Extending Message Handlers with Pattern Matching in the Jadescript Programming Language

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

***Abstract*—Software agents are characterized by sophisticated messaging capabilities that support distributed problem solving and that provide the basic ingredients for interoperability in open agent-based systems. Jadescript is an agent-oriented programming language that has been recently proposed to offer programmers the abstractions that characterize agents to concretely and effectively support the implementation of complex agent-based systems. As expected, the abstractions that Jadescript provides include native support for the advanced messaging capabilities that characterize agents. This paper describes a recent development of Jadescript that extends the language with a native support for pattern matching designed to simplify the reception of structured messages and to ease the management of complex conversations. The proposed support for pattern matching is intimately correlated with the type system of the language, and it can be used to effectively associate inbound messages with specific handlers. From the point of view of programmers, the proposed support for pattern matching allows clearly expressing the intended scope of message handlers, and it contributes to raise the level of abstraction of the language.**

***Index Terms*—Jadescript, JADE, agent-oriented programming, software agents**

CEUR-WS.org/Vol-2404/paper17.pdf

I. Introduction

Since the introduction of software agents (e.g., [1]), the interest in *AOP* (*Agent-Oriented Programming*) (e.g., [2]) has been constantly increasing mainly because AOP is expected to deliver effective tools to design and implement complex agent-based systems. AOP languages shield programmers from many fine-grained details related, for example, to the routing of messages or to the deployment of agents to network hosts. AOP languages promote high-level views of agent-based systems that allow programmers to concentrate on the problems at hand, rather than focusing on fine-grained details that tend to distract attention from problems. All things considered, AOP languages allow programmers to reason on agents at a high level of abstraction because they provide support for the abstractions that characterize the agent-oriented view of software systems promoted, for example, by the IEEE FIPA Standards Committee (www.fipa.org) or by the literature on *AOSE* (*Agent-Oriented Software Engineering*) [3].

The main objective of this paper is to summarize the current state of the development of Jadescript [4], an AOP language whose main purpose is to help programmers to deal with the complexity of building real-world agent-based systems using *JADE* (*Java Agent DEvelopment framework*) (e.g., [5]). In particular, this paper details one of the latest features of the language that has been recently included to ease the reception of structured messages and to support the management of complex conversations among agents. Jadescript now provides a specific support for pattern matching that was carefully integrated with the type system of the language to readily state the conditions used to route messages to proper message handlers. The use of the new support for pattern matching, as described in Section IV, allows programmers to clearly state the intended scope of message handlers, and it ultimately helps to further raise the level of abstraction of the language with respect to previous versions (e.g., [6], [7]).

Even if the early JADE prototypes date back more than twenty years, JADE is still one of the most popular agent platforms [8], and it is still used for academic and industrial projects [8]. In addition, JADE is the solid base for other software platforms like *WADE* (*Workflows and Agents Development Environment*) [9], which supports agent-based business process management, *AMUSE* (*Agent-based Multi-User Social Environment*) [10], which focuses on agent-based multi-player games (e.g., [11]), and *WANTS* (*Workflows and AgeNTS*) [12], which routinely participates in the management of a nation-wide telecommunication network. Jadescript has been recently added to the list of projects that use JADE in order to offer a new way to reduce the complexity of building JADE systems, and to provide concrete support for the adoption of the abstractions that most substantially characterize JADE, namely, agents, behaviours [13], and ontologies [14]. Jadescript is intended to bring the power of JADE to programmers that are not interested in directly using an agent platform, but that are interested in taking advantage of the beneficial features of software agents. Jadescript is meant to enable the effective use of agents as software components (e.g., [15]), and it is already planned that it would allow the adoption of a high-level agent model (e.g., [16]) in the near future.

This paper is organized as follows. Section II provides a brief survey of some of the most relevant languages proposed to support the development of software agents and agent-based systems. Section III describes the major features of Jadescript, and it lists the elements of the language provided to support message passing. Section IV describes the new support for pattern matching. Finally, Section V concludes the paper and outlines major future developments of the discussed research.

II. Related Work

Several AOP languages have been already proposed in the literature, and for some of them, the needed tools, like interpreters and compilers, have also been implemented. Documented experiments on such languages often show that programming languages specifically designed for AOP are convenient for the development of complex agent-based systems. However, most of the languages in use to program software agents and agent-based systems are general-purpose *OOP* (*Object-Oriented Programming*) languages, even if agents and objects differ in important ways, which include, most notably, the degree of autonomy. Objects directly and inevitably invoke methods on other objects, while agents express their intentions to delegate some of their goals to other agents.

AOP languages are based on specific agent models, and they provide linguistic constructs to ease the adoption of such models at a high level of abstraction. Ease of use and expressivity are common characteristics of AOP languages. However, AOP languages differ significantly in terms of the selected agent mental attitudes (if any), of the integration with an agent platform (if any), and of the underlying programming paradigm and implementation language. The literature provides multiple classifications of AOP languages that consider such a diversified landscape. A recent survey [17] proposes a classification of AOP languages that is based on mental attitudes. It identifies the following classes of languages: AOP languages, *BDI* (*Belief-Desire-Intention*) languages, hybrid languages, and other languages. Such a classification acknowledges that BDI languages follow the AOP paradigm, but, for their notable relevance in literature, it reserves special attention to them by listing BDI languages in a separate category. Another appreciated survey [18] proposes a classification in which languages are divided into imperative, declarative, and hybrid. It is worth noting that, in both classifications, the languages that adhere to the declarative programming paradigm are the most numerous because they are natively well-suited to implement automated reasoning. On the contrary, the languages that adopt the imperative programming paradigm are just a minority, and they are frequently obtained by extending procedural programming languages with specific linguistic constructs to support agents and related abstractions. In the rest of this section, some of the most relevant AOP languages are briefly described. Only the languages that share features with Jadescript are considered.

The acronym AOP was first introduced by Shoham [19] together with AGENT-0, a first example of the application of the AOP paradigm. In AGENT-0, a computation is represented by a sequence of collaborative and/or competitive interactions among agents. Each agent has a mental state that is composed of beliefs and commitments. The mental state of an agent changes over time, which is represented as a sequence of discrete steps. Agents share a sense/act cycle in which incoming messages are processed, beliefs and commitments are updated, and actions are executed. An AGENT-0 program is written by enumerating initial states for beliefs and commitments, and by listing commitment rules that refer to future actions. The communication among agents is accomplished by exchanging simple messages, which are classified into three types: inform of a belief, request to perform an action, and unrequest of a previously requested action.

PLAnning Communicating Agents (*PLACA*) [17] is a direct descendant of AGENT-0, and it extends the capabilities of AGENT-0 by providing new linguistic constructs and new mental categories. The major improvement with respect to AGENT-0 is that agents do not need to request for specific actions to call for cooperation, but they can refer to high-level goals. Such an improvement has two major benefits. First, agent communication is lighter because the number of messages is possibly reduced. Second, it allows agents to focus on the desired results of actions, thus easily adding cooperative planning capabilities to agents.

Concurrent MetateM [20] is an AOP language based on temporal logics. In this language, sets of rules are used to describe the lifecycle of agents and the execution of actions. Such rules can be grouped into temporal rules and non-temporal rules. Non-temporal rules are used to support application-specific reasoning, while temporal rules are used to govern agents and they are classified into three categories: start rules, step rules and sometimes rules. In Concurrent MetateM, agents act asynchronously and they interact by message passing. The language does not mandate a structure for messages, and messages are nothing but typed events.

AgentSpeak [21] is an important example of a declarative AOP language. In AgentSpeak, an agent program is described as a tuple that collects beliefs, events (internal and external), actions, plans, and intentions. The approach adopted by AgentSpeak allows to declaratively program agents based on the BDI model. Jason [22] can be considered the first usable implementation of AgentSpeak. Jason is tightly integrated with Java, and it extends AgentSpeak by providing all features needed to effectively adopt it for the implementation of complex agent-based systems. Jason is currently one of the most popular tools to adopt the declarative programming paradigm for the implementation of agent-based systems.

3APL (*An Abstract Agent Programming Language*) [23] is an AOP language that includes abstractions from both declarative and imperative programming paradigms. Agents in 3APL are based on the BDI model and for this reason the language provides a set of abstractions to implement agents with reasoning capabilities. The mental states of agents consist of sets of goals and beliefs, while sets of practical reasoning rules are used to modify mental states and to generate plans to achieve goals. There are two official implementations of the support tools for 3APL, one in Java and one in Haskell.

JACK [24] is an agent platform commercialized by AOS (www.aosgrp.com). It supports the development of agent-based systems composed of agents that are programmed in terms of the BDI model. One of the main elements of the JACK platform is *JAL* (*JACK Agent Language*), an AOP language that is defined as a superset of Java. JAL extends Java by introducing features borrowed from logic languages, and it provides statements to allow the construction of plans. One of the most relevant features of JAL is the native support for organizations and teams, which is provided to enable effective distributed problem solving.

SEA_L (*Semantic web-Enabled Agent Language*) [25], [26] is a *DSL* (*Domain-Specific Language*) to program agent-based systems for the Semantic Web. SEA_L addresses some of the limitations of other development frameworks intended to implement agent-based systems for the Semantic Web. A specific modeling language, called *SEA_ML* (*Semantic web-Enabled Agent Modeling Language*) [27], is available for the graphical modeling of agent-based systems.

CLAIM (*A Computational Language for Autonomous Intelligent and Mobile Agents*) [28] is an AOP language designed with a focus on agent mobility. CLAIM supports holarchies because agents can be built by hierarchically composing other agents. CLAIM agents have two types of reasoning capabilities: forward reasoning, for reactive tasks, and backward reasoning, for goal-driven tasks. Agent communication is performed by message passing, and the underlying agent platform uses a set of specific messages to support agent mobility.

Jadex [29] is a framework to implement agent-based systems originally designed to work on top of JADE. A Jadex agent is equipped with a BDI reasoning engine, and it is programmed in terms of beliefs, goals, and plans. Jadex combines the imperative and the declarative paradigms because it uses *ADFs* (*Agent Definition Files*) to define beliefs, goals and plans, while it uses Java to procedurally define plans. Note that, even if it does not introduce a specific syntax, Jadex underpins an AOP language, and for this reason it is often treated as such. Jadex is intended for practical and commercial use, and a number of real-world applications that use it are documented in the literature. Besides its name, the framework is no longer tightly linked with JADE because now it provides the needed tools to interface various agent platforms.

SARL [30] is an AOP language that follows the imperative programming paradigm. It can be considered as an extension of the Xtend language [31], which is a dialect of Java that it is used to implement the procedural parts of SARL agents. SARL is platform-agnostic, even if it is commonly used with a dedicated agent platform called Janus. One of its most peculiar features is the support for holarchies by means of specific linguistic constructs. SARL compiler is implemented using Xtext [32], which the same development framework used for the Jadescript compiler.

JADEL [33]–[35] is the direct predecessor of Jadescript. It is an AOP language that targets the Java virtual machine, and it is intended to support the construction of agents and agent-based systems using JADE. JADEL provides specific constructs for message passing, for event handling, and for the definition of agents, behaviours, and ontologies. It has direct support for FIPA interaction protocols [36], and its operational semantics is formalized [37]. Finally, its procedural parts are based on the Xtend language, and its major support tools are a compiler and an Eclipse plugin, both built using Xtext.

III. Jadescript in Brief

This paper introduces a new feature of Jadescript intended to embed pattern matching in the core of the language. Such a new feature represents a first attempt at supporting the declarative programming paradigm in Jadescript, and it is designed to make the aims and scope of message handlers explicit. The remaining of this section briefly describes Jadescript and, in particular, it highlights the elements of the language provided to send and receive messages.

Jadescript is a novel programming language designed with the explicit intent to make agent-oriented code similar to semantically-equivalent pseudocode. It supports the development of JADE agents and agent-based systems, and it is characterized by distinctive features designed to make the language very expressive. Notably, the language shares characteristics with popular scripting languages like, for example, the use of semantically-relevant indentation and collection types.

Every Jadescript source code is intended to be compiled into one or more Java source codes, which are then compiled into Java bytecode using any off-the-shelf Java compiler. Such a design choice was taken primarily to grant interoperability with Java, and to enable Jadescript agents to directly use libraries and frameworks already available for the Java virtual machine. Despite the close relationship with Java, Jadescript is not an OOP language, rather it is an AOP language that follows the path originally traced by AGENT-0. The minimal interface to Java, which is still present in Jadescript to support integration with the features of the underlying Java virtual machine, is considered low-level and its use is discouraged.

Jadescript is a statically-typed language, and its type system comprises the following data types:

- Primitive types;
- Collection types;
- Ontology types;
- Agent types; and
- Behaviour types.

Jadescript provides the following primitive types that are immediately mapped to corresponding Java types: `boolean`, `double`, `float`, `integer`, and `text`. It provides collection types in terms of lists and maps of typed values. It supports structured types in terms of ontology types, which can be declared in the scope of `ontology` declarations using `concept`, `action`, `predicate`, and `proposition` declarations. Ontology types are declared (with the exception of `proposition` declarations) in terms of sets of typed properties, and they can include properties inherited from other ontology types. Finally, agent and behaviour types are provided for the manipulation of agents and behaviours. Values of such types cannot be used freely in expressions, and they can be used only in the scope of a limited number of linguistic constructs. The restrictions on the use of agent and behaviour types ensure that programmers cannot freely manipulate agents and behaviours, and they are coherent with the underlying management of the same abstractions in JADE.

In order to improve readability, Jadescript is designed to support a limited form of type inference. The Jadescript compiler can identify when new variables are declared, and it can infer the types of new variables in correspondence of assignment statements. Similarly, the Jadescript compiler can infer the types of properties from the types of expressions used as initializers. Even if the supported form of type inference is sufficient to improve readability, it is worth noting that it is limited with respect to the form of type inference that other languages provide. Actually, the types of some of the elements referenced in source codes like, for example, the types of the formal parameters of procedures, need to be explicitly stated because the language does not provide to the compiler sufficient information to infer them.

Agents are the core abstractions used to build Jadescript agent-based systems, and they depend completely on JADE agents. Each Jadescript agent operates within a JADE container, it is identified by a unique *AID* (*Agent IDentifier*), and it can be in one of several lifecycle states. Agents operate by engaging one or more behaviours. For each agent, active behaviours are executed following the characteristic non-preemptive scheduling mechanisms of JADE behaviours [5]. Currently, Jadescript supports two types of behaviour: `one shot` behaviours and `cyclic` behaviours. Interested readers should consult the official JADE documentation [5] for detailed descriptions of the behaviour scheduling mechanisms and of possible agent-lifecycle states.

The runtime state of agents and behaviours is stored in properties, which are declared using the keyword `property` in the scope of agent and behaviour declarations. Similarly, ontology types are also declared in terms of properties. Properties can be accessed in expressions using the `of` operator, which mimics how the preposition *of* is used in English as a synonym of *belonging to*. Agent and behaviour declarations can also include parameterized blocks of procedural code in the scope of `function` and `procedure` declarations.

Jadescript agents are fully interoperable with JADE agents, and they communicate by exchanging FIPA *ACL* (*Agent Communication Language*) messages. Jadescript provides a set of linguistic constructs to send and receive messages, with message reception expressed in terms of a specific type of event. Jadescript agents can react to events using dedicated linguistic constructs. Future versions of the language are planned to support application-specific types of events, for example, to let agents easily interface with the physical world (e.g. [38]) through the underlying agent container. For the time being, Jadescript supports only three types of events:

- Agent-lifecycle events, handled by the `on create` and the `on destroy` constructs in agent declarations, to allow agents to react to changes of their lifecycle states;
- Behaviour-activation events, handled by the `on create` construct in behaviour declarations, to support the initialization of the internal state of behaviours; and
- Message events, handled by the `on message` construct and its variants in behaviour declarations, to allow the reception of messages with specific characteristics.

```
ontology TemperatureSensor
  proposition nonnegative
  proposition negative
  concept sample(value as double)

cyclic behaviour ReceiveNonNegative
  uses ontology TemperatureSensor

  on inform m
    when content of m is sample do
    s = content of m

    if value of s >= 0 do
      send inform nonnegative
        to sender of m
    else do
      send inform s to aid of agent

cyclic behaviour ReceiveNegative
  uses ontology TemperatureSensor

  on inform m
    when content of m is sample do
    s = content of m

    if value of s < 0 do
      send inform negative
        to sender of m
    else do
      send inform s to aid of agent
```

Fig. 1. Example of the limitations of the traditional approach used to associate incoming messages to appropriate message handlers in Jadescript.

Event handlers, just like functions and procedures, have a body where procedural code is included by means of statements and expressions. Moreover, message handlers can specify a `when` clause that is used to state a condition that interesting messages are required to satisfy. In detail, Jadescript allows the use of an expression after the optional keyword `when` in the declaration of a message handler to allow programmers to state a condition that messages must satisfy in order to be extracted from the message queue of the agent. Such expressions are not arbitrary Boolean expressions, but they are conditions, which can be composed using the ordinary logical connectives, on the performative, the ontology, and the type of the content of messages. For the sake of readability, a condition on the performative of messages can also be declared by stating the accepted performative after the keyword `on` so that, for example, `on inform` can be used to declare a message handler that processes inform messages. Fig. 1 shows an example of the use of the discussed mechanism to route inbound messages to appropriate message handlers. Note that the described mechanism also provides valuable compile-time information about handled messages that is used to infer types.

IV. Pattern Matching in Message Handlers

The urge to extend Jadescript with an improved mechanism to declare message handlers is mainly motivated by the need to provide more expressive linguistic constructs to associate inbound messages with appropriate handlers. In particular, the major weakness of the described support for message dispatching becomes evident when several behaviours work on similar messages. Consider, for example, the behaviours shown in Fig. 1. If the two behaviours are activated by the same agent, then they would work on the same message queue, and they would handle messages with the same ontology, the same performative, and the same content type. However, behavior `ReceiveNonNegative` is designed to handle messages for which the property `value` is nonnegative, while behaviour `ReceiveNegative` has the opposite requirement on the same property. In this example, both behaviours reinsert the received message into the message queue of the agent when the message does not meet the intended requirements. Even if the reinsertion into the message queue ensures that messages are eventually processed by appropriated handlers, the example emphasizes that the current support for message dispatching is probably too limited.

A simple solution to the mentioned weakness of the support for message dispatching that Jadescript has been providing since its early releases could be based on the possibility of allowing generic Boolean expressions to guard the activation of handlers. However, such a solution is not fully satisfactory because generic Boolean expressions could have uncontrollable side effects. This is the reason why Jadescript has been recently extended to allow the use of pattern matching in the guard expressions of message handlers. In particular, the new support for pattern matching in message handlers is made available to programmers by means of a new binary operator called `matches` that allows to compare a value against a specified pattern. Such an operator evaluates to `true` when the value matches to the provided pattern, and it evaluates to `false` otherwise. Currently, Jadescript supports the following four types of patterns:

- Ontology patterns, used to check a value of an ontology type against a pattern;
- List patterns, used to check a list against a pattern;
- Map patterns, used to check a map against a pattern; and
- Regular-expression patterns, used to check if a text satisfies a regular expression.

Ontology, list, and map patterns are called composite patterns because they are defined as sequences of terms. Each term in a composite pattern can be:

- A `text`, `integer`, `float`, `double`, or `boolean` literal;
- A variable identifier, possibly not yet declared in the current scope;
- An underscore placeholder symbol (_), which acts as a dummy variable; or
- A pattern, which acts as a sub-pattern.

```
cyclic behaviour ReceiveNonNegative
  uses ontology TemperatureSensor

  on inform m
    when
      content of m matches sample(v) and
      v >= 0 do
    send inform nonnegative
      to sender of m

cyclic behaviour ReceiveNegative
  uses ontology TemperatureSensor

  on inform m
    when
      content of m matches sample(v) and
      v < 0 do
    send inform negative
      to sender of m
```

Fig. 2. Example of message handlers that use the `matches` operator with appropriate ontology patterns.

The pattern matching mechanism for composite patterns works by comparing each term of the pattern, from left to right, to the corresponding value at the left-hand side of the `matches` operator. In the case of a literal term, the value represented by the literal is simply compared for equality. The same sort of comparison is performed for identifier terms that refer to variables already present in the current scope. However, if the pattern refers to an identifier that cannot be resolved to a declared variable in the current scope, the `matches` operator binds the identifier to the corresponding values in the left-hand side operand. Such bindings are then treated as if they were declared variables. In particular, when the checking of a pattern results in the implicit declaration of a variable, such a new variable becomes accessible in the current scope. Similarly, if the term to be checked is an underscore placeholder, any value is considered to be a valid match because the pattern is treated as a dummy variable. Finally, when the term to be checked is a sub-pattern, the described matching mechanism is performed recursively. Fig. 2 shows an example of the use of ontology patterns.

List and map patterns are similar to list and map literals, with the addition of the possibility of using the Prolog-inspired optional pipe symbol to separate the head from the tail (also known as rest) of the collection. Note that the optional pipe symbol is particularly relevant because it is used to express patterns that match lists and maps of unknown size.

Regular expression patterns, as the name suggests, are regular expressions for strings of characters. The `matches` operator checks if the value at the left-hand side of the operator is a text, and if it is, returns `true` if the text matches the pattern. The syntax of this type of patterns is inspired from the syntax of regular-expression literals in Javascript.

V. Conclusion

This paper described a recent development of Jadescript that extends the language with a native support for pattern matching. The proposed support for pattern matching is intimately related with the type system of the language, and it can be used to effectively associate inbound messages with specific handlers. From the point of view of programmers, such a new feature of the language allows clearly expressing the intended aims and scope of message handlers, and it ultimately contributes to raise the level of abstraction of the language.

References

[1] J. Bradshaw, *Software Agents*. MIT Press, 1997.

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

[3] F. Bergenti, M.-P. Gleizes, and F. Zambonelli, Eds., *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*. Springer, 2004.

[4] F. Bergenti, S. Monica, and G. Petrosino, "A scripting language for practical agent-oriented programming," in *Proc. 8^{th} ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018) at ACM SIGPLAN Conference Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2018)*. ACM Press, 2018, pp. 62–71.

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

[6] F. Bergenti and G. Petrosino, "Overview of a scripting language for JADE-based multi-agent systems," in *Proc. 19^{th} Workshop "From Objects to Agents" (WOA 2018)*, ser. CEUR Workshop Proceedings, vol. 2215. RWTH Aachen, 2018, pp. 57–62.

[7] G. Petrosino and F. Bergenti, "An introduction to the major features of a scripting language for JADE agents," in *Proc. 17^{th} Conference of the Italian Association for Artificial Intelligence (AI*IA 2018)*, ser. Lecture Notes in Artificial Intelligence, vol. 11298. Springer, 2018, pp. 3–14.

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

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

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

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

[12] 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.

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

[14] M. Tomaiuolo, P. Turci, F. Bergenti, and A. Poggi, "An ontology support for semantic aware agents," in *Proc. International Workshop on Agent-Oriented Information Systems (*AOIS 2005*)*, ser. LNAI, vol. 3529. Springer, 2006, pp. 140–153.

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

[16] F. Bergenti and A. Poggi, "A development toolkit to realize autonomous and inter-operable agents," in *Proc.* 5^{th} *International Conference on Autonomous Agents*, 2001, pp. 632–639.

[17] C. Bădică, Z. Budimac, H.-D. Burkhard, and M. Ivanovic, "Software agents: Languages, tools, platforms," *Computer Science and Information Systems*, vol. 8, no. 2, pp. 255–298, 2011.

[18] R. H. Bordini, L. Braubach, M. Dastani, A. El Fallah Seghrouchni, J. J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci, "A survey of programming languages and platforms for multi-agent systems," *Informatica*, vol. 30, no. 1, 2006.

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

[20] M. Fisher, "A survey of concurrent MetateM – The language and its applications," in *Temporal Logic*. Springer, 1994, pp. 480–505.

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

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

[23] K. V. Hindriks, F. S. De Boer, W. Van der Hoek, and J.-J. C. Meyer, "Agent programming in 3APL," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 4, pp. 357–401, 1999.

[24] M. Winikoff, "JACK intelligent agents: An industrial strength platform," in *Multi-Agent Programming*. Springer, 2005, pp. 175–193.

[25] S. Demirkol, M. Challenger, S. Getir, T. Kosar, G. Kardas, and M. Mernik, "SEA_L: A domain-specific language for Semantic Web enabled multi-agent systems," in *Proc. Federated Conference on Computer Science and Information Systems (FedCSIS 2012)*, 2012, pp. 1373–1380.

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

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

[28] A. El Fallah-Seghrouchni and A. Suna, "Claim: A computational language for autonomous, intelligent and mobile agents," in *Proc. International Workshop Programming Multi-Agent Systems (ProMAS 2003)*. Springer, 2003, pp. 90–110.

[29] L. Braubach, A. Pokahr, and W. Lamersdorf, "Jadex: A BDI-agent system combining middleware and reasoning," in *Software Agent-Based Applications, Platforms and Development Kits*, R. Unland, M. Calisti, and M. Klusch, Eds. Birkhäuser, 2005, pp. 143–168.

[30] S. Rodriguez, N. Gaud, and S. Galland, "SARL: A general-purpose agent-oriented programming language," in *Proc. IEEE/WIC/ACM International Joint Conferences of Web Intelligence (WI 2014) and Intelligent Agent Technologies (IAT 2014)*, vol. 3. IEEE, 2014, pp. 103–110.

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

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

[33] F. Bergenti, "An introduction to the JADEL programming language," in *Proc. IEEE 26^{th} International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE Press, 2014, pp. 974–978.

[34] 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 International Publishing, 2016, pp. 213–224.

[35] 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.

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

[37] 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.

[38] 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.