Two Agent-Oriented Programming Approaches Checked Against a Coordination Problem

Eleonora Iotti, Giuseppe Petrosino, Stefania Monica, and Federico Bergenti(✉)

Dipartimento di Scienze Matematiche, Fisiche e Informatiche,
Università degli Studi di Parma, Parma, Italy
{eleonora.iotti,stefania.monica,federico.bergenti}@unipr.it,
giuseppe.petrosino@studenti.unipr.it

Abstract. This paper discusses two approaches to agent-oriented programming and compares them from a practical point of view. The first approach is exemplified by Jadescript, which is an agent-oriented programming language that has been recently proposed to simplify the adoption of JADE. The second approach is exemplified by Jason, which is currently one of the most popular agent-oriented programming languages. Jason can be configured to use JADE to support the distribution of agents, which ensures that the discussed comparison between the two approaches can also take into account the performance of implemented multi-agent systems. In order to devise a quantitative comparison, the two considered languages are used to solve the same coordination problem, and obtained implementations are compared to discuss advantages and drawbacks of the two approaches.

Keywords: Jadescript · Jason · Agent-oriented programming

1 Introduction

The development of *AOP* (*Agent-Oriented Programming*) languages (e.g., [23]) and the study of *AOSE* (*Agent-Oriented Software Engineering*) (e.g., [6]) are of primary importance for the community of researchers and practitioners interested in software agents and agent-based software development. In the last few years, a plethora of methodologies, languages, and tools were presented in the literature (e.g., [18,19]). AOP languages are generally recognized in such a body of literature as important tools for the development of agent technologies, in contrast to traditional (lower-level) languages, which are often considered (e.g., [1,10]) not suitable to effectively implement software agents and agent-based software systems with the desired characteristics (e.g., [3]).

One of the approaches to AOP that attracted much attention is the *BDI* (*Belief-Desire-Intention*) approach [22]. In brief, the BDI approach schematizes software agents in terms of human-like features, and it targets the design of agents as intelligent software entities that are able to plan how to effectively bring about their intentions on the basis of their desires and their beliefs. The BDI approach is actually implemented in several frameworks and languages, which include Jason [12], *GOAL* (*Goal-Oriented Agent Language*) [16], *JAL* (*JACK Agent Language*) [25], and *3APL* (*An Abstract Agent Programming Language*) [15]. Note that the BDI approach is inherently declarative, and its various implementations are mostly derived from logic programming. The tight connection between the BDI approach and logic programming is often considered as a limitation for the popularization of the BDI approach because logic programming languages tend to have a steep learning curve for the average programmer. On the one hand, experts of agent technologies are comfortable with the BDI approach, but on the other hand, students and practitioners that try to adopt the BDI approach tend to be discouraged. This is the reason why the literature has recently witnessed the introduction of AOP languages that follow procedural approaches, or that mix declarative features with procedural features to obtain sophisticated hybrid languages (e.g., [13,14]).

In addition to AOP languages, several software frameworks are available to support the construction of agent-based software systems. As stated in a recent survey [18], among such frameworks, the primacy in terms of popularity is held by *JADE* (*Java Agent DEvelopment framework*) (e.g., [2]) in both the academia and the industry. JADE provides a rich programming interface for developers who wants to program software agents, and it also provides a runtime platform to execute *MASs* (*Multi-Agent Systems*) on distributed computing architectures. Unfortunately, the general perception is that a sophisticated tool like JADE is hard to learn for the newcomers to agent-based software development (e.g., [7]) because it requires advanced skills in both object-oriented and agent-oriented programming. *JADEL* (*JADE Language*) (e.g., [4,7]) is a recent *DSL* (*Domain-Specific Language*) that aims at simplifying the use of JADE by embedding in the language the view of agents and MASs that JADE proposes. Jadescript [8,9,20,21] leverages the experience on JADEL to propose a new AOP language designed to simplify the construction of agents and MASs. Jadescript is an entirely new language that provides linguistic constructs designed to concretize the abstractions that characterize agent-based software development in the attempt to promote their effective use.

The major contribution of this paper is to assess some of the characteristics of Jadescript by means of a comparison with another AOP language in terms of the effectiveness in the construction of a solution to an illustrative problem. Jason was chosen as the point of reference to assess the characteristics of Jadescript, and such a choice offers the opportunity to compare the procedural approach to AOP that Jadescript advocates with the BDI approach that Jason uses. In particular, Sect. 2 briefly describes the major features that Jadescript and Jason offer to the programmer to master the complexity of agent-based software development. Section 3 describes the illustrative problem chosen to compare the two languages. Section 4 reports on a quantitative assessment of the implemented solutions to the studied problem in terms of suitable metrics, and it also compares the two languages in terms of a subset of the criteria described in [11], which are accepted criteria for the comparison of AOP languages. Finally, Sect. 5 concludes the paper by summarizing major documented results.

2 Jadescript and Jason in Brief

This section summarizes the major features that Jadescript and Jason offer, and it highlights their commonalities and differences. Only the features that are relevant for the comparison discussed in Sect. 4 are presented.

Linguistic Features. Jason is a powerful implementation of AgentSpeak, and therefore it follows a declarative approach to AOP. The syntax that Jason uses is very similar to the syntax originally proposed for AgentSpeak. Beliefs, goals, intentions, and plans are specified for each agent in the MAS. Jason agents are immersed in an environment, which is defined explicitly by naming a MAS together with its infrastructure and agents.

Jadescript is grounded on JADE, and therefore it supports the construction of agents that execute in the scope of the containers of a regular JADE platform. A Jadescript agent has a list of behaviours to perform tasks, and behaviours are defined procedurally to handle messages and to react to events. Following a procedural approach to AOP, Jadescript allows the declaration of behaviours in terms of properties, procedures, functions, message handlers, and event handlers. In addition, Jadescript allows the declaration of ontologies to support the communication among agents. Ontologies are declared in Jadescript in terms of propositions, predicates, concepts, and actions.

In summary, Jason is a declarative language, while Jadescript has declarative features but it is essentially a procedural language. The declarative features that Jadescript provide are meant to support pattern matching and event-driven programming, but the actions that agents perform are defined procedurally.

Multi-agent Systems. Jason provides a clear way to define MASs. The name of the MAS must always be specified, and its infrastructure must be chosen. The most common infrastructures are immediately available, and they are called `Centralized` and `Jade`. When the `Centralized` infrastructure is used, all Jason agents execute within the same process. On the contrary, the `Jade` infrastructure is based on JADE, and agents can be easily distributed across JADE containers when such an infrastructure is used. In the Jason declaration of a MAS, a Java class that defines the actual environment can be referenced by means of the keyword `environment`. Finally, the most important part of the definition of a MAS in Jason is the list of the agents that populate the environment. Such agents have names and references to specific agent files for their definitions.

Currently, Jadescript does not provide a specific support for the construction of a MAS because it assumes that JADE is used to activate agents within the environment. Jadescript agents are launched using JADE, and the infrastructure is managed in terms of a regular JADE platform. In Jadescript, the knowledge about the environment is passed among agents through messages, and ontologies are essential in such an exchange of messages.

In summary, a relevant difference between Jadescript and Jason is in the support for the construction of MASs. Actually, Jadescript does not yet offer a means to construct a MAS, but it assumes that JADE is directly used.

Agents. The declaration of a Jason MAS references a set of agent files, which declaratively define for each agent the initial set of beliefs (and relative rules), the initial set of goals (and relative rules), and all available plans. The declaration of a Jadescript agent consists of a set of properties and a set of lifecycle event handlers. Lifecycle event handlers are used to programmatically define what an agent should do to react to the changes of its lifecycle state.

The approaches adopted by Jason and by Jadescript for the declaration of agents are strongly related. They declare agents in terms of internal beliefs or properties, supported languages or ontologies, and available behaviours or plans. On the contrary, the syntaxes adopted by the two languages are very different. Jason is based on self-contained agent files, and the rules for beliefs and goals are written using a Prolog-like notation. Jadescript keeps a small part of the code of an agent in the agent declaration because behaviours, ontologies, functions, and procedures are coded outside of the agent declaration to promote reusability. Similarly, Jadescript supports inheritance to promote reusability by providing the programmer with the ability to extend the definitions of agents, behaviours, and ontologies. Instead, Jason does not offer inheritance and related features.

Plans and Behaviours. Plans are basic courses of actions for Jason agents, and a plan in Jason consists of a triggering event, a context, which is a set of actual beliefs, and a body. The body of a plan includes the actions to be performed and the sub-goals to be achieved. A goal in Jason is strictly related to the environment because it describes a state of the environment that the agent wants to actualize.

Behaviours are the means that Jadescript offers to define the tasks that agents perform. Each agent can activate multiple behaviours whose order of execution during the agent lifecycle is not explicitly declared by the programmer. Activated behaviours are instead collected into an internal list, and their execution is autonomously scheduled by the agent with using a non-preemptive scheduler. Note that a behaviour can directly access the internal state of the agent that activated it by accessing its properties, functions, and procedures.

Jadescript behaviours can be matched with Jason plans, but Jason plans are declarative while Jadescript behaviours are procedural. In particular, Jadescript behaviours are not explicitly related to the goals that they help to achieve.

Events. Events in Jason are related to the beliefs and the goals of each agent. Actually, an event can add or delete beliefs and goals, and it can also trigger the activation of a plan to bring about a goal.

Jadescript behaviours normally include event handlers, and event handlers are executed when managed events occur. One of the most recent additions to the language is the support for pattern matching [21], which allows declaratively defining the structure of managed events for each event handler.

Both languages support event handlers in terms of reactions to events. In both cases, the activation of event handlers affects the states of agents. The main difference between Jadescript and Jason with respect to event handlers is the presence of goals, which are explicit in Jason and implicit in Jadescript. Finally, it is worth noting that in both languages events can be internal or external.

Ontologies. Ontologies are of primary importance in Jadescript because they are used to support communication, and all non-trivial Jadescript agents are expected to reference at least one ontology. Actually, ontologies are used in Jadescript to state a fact or a relation about elements of the domain of the problem. On the contrary, Jason does not provide a support for ontologies. Note that ontologies cannot be reduced to beliefs because ontologies are descriptions of the domain of the problem, and events can neither add nor remove ontologies.

3 The Santa Claus Coordination Problem

The Santa Claus coordination problem [24] is a well-known coordination problem that is expressed, in its simplest form, as follows. Santa Claus has nine reindeer and ten elves in its team. He sleeps waiting for a group formed by all of his reindeer or by three of his elves. When three elves are ready and awake Santa Claus, he must work with them on new toys. Similarly, when the group of reindeer is ready and awakes Santa Claus, they work together to deliver toys. It is important that Santa Claus is awakened only when a group with the correct size is formed. In addition, Santa Claus should give priority to the reindeer in case that there are both a group of elves and a group of reindeer ready.

The major reasons for choosing the Santa Claus coordination problem for the comparison between Jadescript and Jason can be listed as follows. First, the problem is simple, but not trivial, and a number of solutions are documented in the literature. In particular, one of such solutions has already been implemented in JADEL [17], and the Jadescript solution used for the experiments described in Sect. 4 is a rework of such an implementation. Second, a Jason implementation can be found in the official Jason distribution[1], and such an implementation was used for the experiments described in Sect. 4. Third, the problem depends on numeric parameters that can be used to vary the characteristic size of the problem and to support quantitative comparisons.

In order to support a fair comparison among the mentioned solutions in Jadescript and Jason, the architecture adopted in the Jason solution was used to design the Jadescript solution. In particular, the architecture of both solutions consists of Santa Claus, nineteen workers (ten elves and nine reindeer), and two secretaries to schedule appointments with workers. Secretaries are called Edna (assigned to the elves) and Robin (assigned to the reindeer), and their responsibility is to form groups and to awake Santa Claus when needed.

[1] Version 2.4 downloaded from the official site (http://jason.sourceforge.net).

The remaining of this section describes the Jadescript solution to the Santa Claus coordination problem. The Jadescript solution uses four types of message contents, as follows. The `workerReady` proposition is used by workers to inform their secretary that they are ready, and the secretary, in turn, uses the `groupFormed` proposition to inform Santa Claus that a group is formed. The `OK` proposition is used by Santa Claus to notify the beginning of the working phase to the chosen group of workers. Finally, the `done` proposition is used by workers to tell that they have completed their jobs.

In the Jadescript solution, the actions of elves and reindeer are defined by means of two behaviours, which are activated by workers at startup, as shown in Fig. 1 (lines 8 and 9). The behaviour `SendReady` is declared (line 11) as `one shot`, and, as such, it contains a single action defined by the procedural code that follows the keyword `do`. When a Jadescript agent activates a behaviour, the behaviour is added to the list of active behaviours of the agent, and then the agent tries to execute it using its non-preemptive scheduler. A `one shot` behaviour is removed from the list of active behaviours just after its execution, and therefore the `SendReady` behaviour is used just once to send a `workerReady` message to the appropriate secretary (line 12). On the contrary, the `WaitForOK` behaviour is declared as `cyclic` (line 14). Cyclic behaviours do not leave the list of the active behaviours of the agent until they are explicitly deactivated, and therefore they are suitable for cyclic tasks such as message handling. In particular, the Jadescript construct that begins with the keywords `on inform` (line 15) declares a message handler that is executed at the reception of a message with performative `inform` and content `OK`. When a message with such characteristics is received, the worker first starts its work (lines 17 and 19), then it replies with a `done` message to Santa Claus (line 20), and it finally sends a `workerReady` message to its secretary (line 21).

The remaining of this section discusses the behaviours used by the Jadescript solution to the Santa Claus coordination problem to implement the two secretaries and Santa Claus. The Jadescript source codes of agents and behaviours is not presented in the paper due to page restrictions.

In the adopted architecture, secretaries count how many workers are actually ready, and when a group of the needed size is available, they promptly inform Santa Claus using a specific message. When created, a secretary activates only one cyclic behaviour, namely `HandleWorkerMessages`, and it shares three properties, namely `groupSize`, `kind`, and `santa`, with the behaviour. When scheduled for execution, the behaviour first checks for `workerReady` messages from workers, then it collects the identifiers of the senders, and it finally sends a `groupFormed` message to Santa Claus. Note that the reindeer communicate exclusively with Robin while elves communicate exclusively with Edna, so the group formation processes are handled independently for the two groups.

Santa Claus uses two behaviours to implement its two possible states. The first behaviour, which corresponds to the first state, is the cyclic behaviour `WaitForGroups`. Such a behaviour first checks if there is a message in the queue that states that a group of reindeer is ready. If no groups of reindeer are ready, the behaviour performs a similar check for a group of elves. When a group is correctly formed, Santa Claus first sends an `OK` message to all the workers in the group, and then it changes its state by deactivating the `WaitForGroups` behaviour to activate the `WaitJobCompletion` behaviour. The `WaitJobCompletion` behaviour is used to define the actions to perform when Santa Claus is in the second state. In such a state, Santa Claus waits for all the workers to send `done` messages, then the `WaitJobCompletion` behaviour is deactivated in favor of the `WaitForGroups` behaviour to perform a transition back to the first state.

```
agent Worker uses ontology SantaClausProblem
  property secretary as aid
  property myKind as text

  on create with kind as text, secretaryName as text do
    secretary = aid(secretaryName)
    myKind = kind
    activate behaviour WaitForOK
    activate behaviour SendReady

  one shot behaviour SendReady do
    send inform workerReady to secretary

  cyclic behaviour WaitForOK
    on inform when content is OK do
      if myKind = "reindeer" do
        do sleep with millis = 3000+rand(1000)
      else do
        do sleep with millis = 1000+rand(1000)
      send inform done to sender of message
      activate behaviour SendReady
```

Fig. 1. The `Worker` agent and its `SendReady` and `WaitForOK` behaviours in Jadescript.

4 Quantitative Comparison

The quantitative comparison between Jadescript and Jason presented in this section is based on three solutions to the Santa Claus coordination problem. The considered implementations are: the Jadescript implementation presented in Sect. 3, the Jason implementation on the `Centralized` infrastructure, and the Jason implementation on the `Jade` infrastructure. Note that the use of JADE allows comparing execution times. Also note that the execution time of the Jason implementation on the `Centralized` infrastructure is reported as baseline.

In order to unambiguously measure execution times, only Santa Claus, Robin, and the reindeer were activated. Hence, the settings for a single execution are described in terms of the number of reindeer R and the number of works to be done W. Under such an assumption, the execution times measure how well the implementations can handle the increase of the number of agents, and how well agents are capable of handling message exchanges. Two types of experiments were performed: in the first type, execution times were measured when the number of works is kept low and the number of reindeer increases, while in the second type, execution times were measured when the number of reindeer

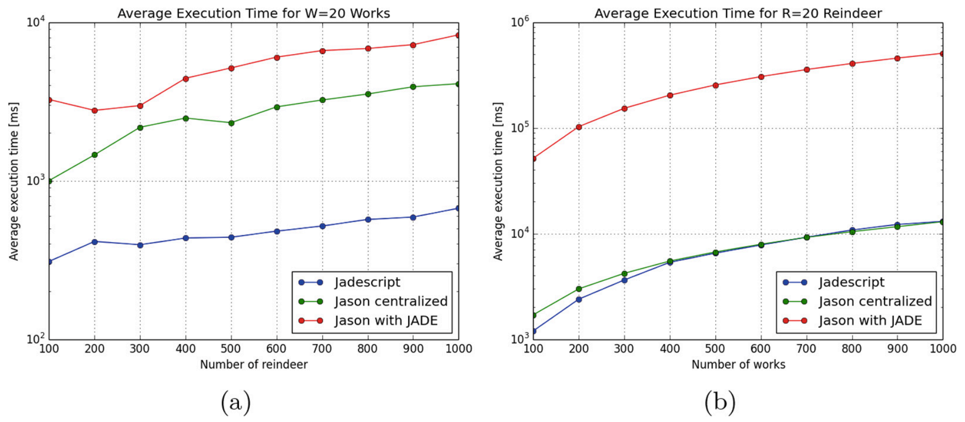

Fig. 2. Plots of execution times for the three experiments as (a) $W = 20$ and the number of reindeer increases, and (b) $R = 20$ and the number of works increases.

is kept low and the number of works increases. For the first type of experiment, a fixed number of $W = 20$ works is chosen, and reindeer range from $R = 100$ to $R = 1000$. For the second type of experiment, only $R = 20$ reindeer are considered, but the works to be done range from $W = 100$ to $W = 1000$. For all considered executions, the termination condition was expressed in terms of the number of works done.

All experiments were repeated for 100 iterations for each considered configuration, and the average execution times over the 100 iterations were recorded. For all experimented configurations, the measurement of execution times started at the activation of Santa Claus, and it stopped when Santa Claus worked W times with reindeer and a group of reindeer was formed for the $(W+1)$–th time. Such a choice allowed to ignore the time needed to start up and shut down the platform, so that only the actual execution times of agents were recored. The experiments were executed on an Apple MacBook Air mid-2013 with a 1.3 GHz Intel Core i5, 3 MB L3 shared cache, 4 GB LPDDR3 1600 MHz RAM, Java version 12, Jason version 2.4, and JADE version 4.3.0.

The execution times for all experiments are shown in Fig. 2. When the number of reindeer increases and $W = 20$, the Jadescript solution is much faster than the Jason solutions, as shown in Fig. 2(a). When the number of works increases and $R = 20$, the performance of the Jason solution with the `Centralized` infrastructure is very similar to the performance of the Jadescript solution, as shown in Fig. 2(b). On the contrary, Fig. 2(b) shows that the Jason solution with the `Jade` infrastructure is more than ten time slower that the Jadescript solution.

Even if the measured execution times shown in Fig. 2 offer an interesting point of view to compare the two considered languages, other criteria must be taken into account to propose a fair comparison. For example, a comparison between the Jadescript solution and the Jason solutions in terms of *LOCs* (*Lines Of Code*) shows that Jason solutions require only 45 LOCs instead of 71 LOCs.

Besides LOCs, it is worth noting that [11] enumerates a list of accepted criteria to evaluate tools for agent-based software development. Only a few of the criteria proposed in [11] are applicable to support a comparison between Jadescript and Jason. Criterion 1(d) in [11] focuses on the simplicity of AOP tools. In this perspective, the Jason solutions are surely elegant and they require less LOCs. On the other hand, Jadescript is very close to an agent-oriented pseudocode and this is the reason why the Jadescript solution looks simpler and more readable. Also, Jason heavily relies on operators instead of keywords, which makes the Jadescript solution definitely more understandable than the Jason solutions, especially for the newcomers to AOP. Finally, criterion 1(i) in [11] emphasizes the relevance of the support for software engineering principles. Some well-known AOSE methodologies for the BDI approach are applicable to Jason, but they are not directly supported by the language. The Jadescript solution, instead, could be enhanced by means of available constructs in the language for inheritance (e.g., by specializing `Worker` agents as `Reindeer` agents and `Elf` agents) and modularization (e.g., by moving behaviour declarations outside of agent declarations) to better support maintainability, composability, and reusability.

5 Conclusion

Jadescript has been recently introduced as an AOP language that targets the complexity of building agent-based software systems with JADE. A few examples of the use of Jadescript have already been proposed [8,9,20,21] to present the language to the community of researchers and practitioners interested in agent-based software development, but a quantitative comparison with another AOP language was lacking. This paper compares Jadescript and Jason on a specific problem, but comparisons with other languages and on other problems have already been planned for the near future.

As far as performance is concerned, Fig. 2(a) shows that Jadescript helps the programmer in fine-tuning the performance of the MAS, and Fig. 2(b) shows that Jadescript ensures an effective use of JADE. Therefore, from the reported performance comparison, it is evident that Jadescript is preferable to Jason for the construction of agent-based software systems intended to scale to real-world applications (e.g. [5]) with a large number of agents.

References

1. Bădică, C., Budimac, Z., Burkhard, H.D., Ivanovic, M.: Software agents: languages, tools, platforms. Comput. Sci. Inf. Syst. **8**(2), 255–298 (2011)
2. Bellifemine, F., Bergenti, F., Caire, G., Poggi, A.: JADE – A Java Agent DEvelopment framework. In: Multi-Agent Programming, pp. 125–147. Springer (2005)
3. Bergenti, F.: A discussion of two major benefits of using agents in software development. In: Petta, P., Tolksdorf, R., Zambonelli, F. (eds.) Engineering Societies in the Agents World III. Lecture Notes in Artificial Intelligence, vol. 2577, pp. 1–12. Springer, Heidelberg (2003)

4. Bergenti, F.: An introduction to the JADEL programming language. In: Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2014), pp. 974–978. IEEE (2014)
5. Bergenti, F., Caire, G., Gotta, D.: Large-scale network and service management with WANTS. In: Industrial Agents: Emerging Applications of Software Agents in Industry, pp. 231–246. Elsevier (2015)
6. Bergenti, F., Gleizes, M.P., Zambonelli, F. (eds.): Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. Springer, Boston (2004)
7. Bergenti, F., Iotti, E., Monica, S., Poggi, A.: Agent-oriented model-driven development for JADE with the JADEL programming language. Comput. Lang. Syst. Struct. **50**, 142–158 (2017)
8. Bergenti, F., Monica, S., Petrosino, G.: A scripting language for practical agent-oriented programming. In: Proceedings of the 8th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018), pp. 62–71. ACM (2018)
9. Bergenti, F., Petrosino, G.: Overview of a scripting language for JADE-based multi-agent systems. In: Proceedings of the 19th Workshop "From Objects to Agents" (WOA 2018). CEUR Workshop Proceedings, vol. 2215, pp. 57–62 (2018)
10. Bordini, R.H., Braubach, L., Dastani, M., El Fallah Seghrouchni, A., Gomez-Sanz, J.J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. Informatica **30**(1) (2006)
11. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A.: Multi-Agent Programming. Springer, Boston (2005)
12. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-agent Systems in AgentSpeak Using Jason. Wiley, Hoboken (2007)
13. Challenger, M., Mernik, M., Kardas, G., Kosar, T.: Declarative specifications for the development of multi-agent systems. Comput. Stand. Inter. **43**, 91–115 (2016)
14. Fichera, L., Messina, F., Pappalardo, G., Santoro, C.: A Python framework for programming autonomous robots using a declarative approach. Sci. Comput. Program. **139**, 36–55 (2017)
15. Hindriks, K.V., De Boer, F.S., Van der Hoek, W., Meyer, J.J.: Agent programming in 3APL. Auton. Agent. Multi Agent Syst. **2**(4), 357–401 (1999)
16. Hindriks, K.V., Dix, J.: GOAL: a multi-agent programming language applied to an exploration game. In: Shehory, O., Sturm, A. (eds.) Agent-Oriented Software Engineering, pp. 235–258. Springer, Heidelberg (2014)
17. Iotti, E., Bergenti, F., Poggi, A.: An illustrative example of the JADEL programming language. In: Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART 2018), vol. 1, pp. 282–289. ScitePress (2018)
18. Kravari, K., Bassiliades, N.: A survey of agent platforms. J. Artif. Soc. Soc. Simul. **18**(1), 11 (2015)
19. Müller, J.P., Fischer, K.: Application impact of multi-agent systems and technologies: a survey. In: Shehory, O., Sturm, A. (eds.) Agent-Oriented Software Engineering, pp. 27–53. Springer, Heidelberg (2014)
20. Petrosino, G., Bergenti, F.: An introduction to the major features of a scripting language for JADE agents. In: Ghidini, C., Magnini, B., Passerini, A., Traverso, P. (eds.) Advances in Artificial Intelligence (AI*IA 2018), pp. 3–14. Springer, Cham (2018)

21. Petrosino, G., Bergenti, F.: Extending message handlers with pattern matching in the Jadescript programming language. In: Proceedings of the 20th Workshop "From Objects to Agents" (WOA 2019). CEUR Workshop Proceedings, vol. 2404, pp. 113–118 (2019)
22. Rao, A.S., Georgeff, M.P.: BDI agents: From theory to practice. In: Proceedings of the 1st International Conference on Multiagent Systems (ICMAS 1995), vol. 95, pp. 312–319. AAAI (1995)
23. Shoham, Y.: An overview of agent-oriented programming. In: Software Agents, vol. 4, pp. 271–290. MIT Press (1997)
24. Trono, J.A.: A new exercise in concurrency. ACM SIGCSE Bull. **26**(3), 8–10 (1994)
25. Winikoff, M.: JACK intelligent agents: an industrial strength platform. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah, Seghrouchni A. (eds.) Multi-Agent Programming, pp. 175–193. Springer, Boston (2005)