Imperative and Event-Driven Programming of Interoperable Software Agents

Giuseppe Petrosino[1,✉], Stefania Monica[1], and Federico Bergenti[2]

[1] Dipartimento di Scienze e Metodi dell'Ingegneria
Università degli Studi di Modena e Reggio Emilia, Italy
{giuseppe.petrosino,stefania.monica}@unimore.it
[2] Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università degli Studi di Parma, Italy
federico.bergenti@unipr.it

Abstract. Jadescript is a recent agent-oriented programming language conceived to support the effective development of agents and multi-agent systems based on JADE. Jadescript is designed to ease the development of agents by means of a tailored syntax matched with a programmer-friendly development environment. This paper presents a brief overview of Jadescript by describing its main features and abstractions and by comparing them with the corresponding features and abstractions advocated by other agent-oriented programming languages. Moreover, to show the applicability of Jadescript to the construction of interesting multi-agent systems, this paper concisely summarizes a case study in which Jadescript is used to implement agents that participate in English auctions. Finally, this paper is concluded with a brief overview of planned future developments of the language.

Keywords: Agent-oriented software engineering · Agent-oriented programming · Jadescript · JADE

1 Introduction

Over the past two decades [7], numerous researchers and practitioners have effectively used *JADE* (*Java Agent DEvelopment framework*) [2] for their projects. By taking the role of the reference implementation of *FIPA* (*Foundation for Intelligent Physical Agents*) [35] specifications, JADE has significantly contributed to shape the ideas, the methodologies, and the tools of *AOSE* (*Agent-Oriented Software Engineering*) [9]. In particular, JADE helped promote a peculiar view of agents that focuses on the features of agents that are considered as useful for software development. Essentially, in this view, agents are software components [4] that engage in complex interactions [24] by exchanging messages in possibly heterogeneous [1,15] and challenging [8,12,16,21,23] environments.

Recent trends in software engineering, exemplified by *DSLs* (*Domain-Specific Languages*) [22], suggested that the adoption of the peculiar view of agents that JADE advocates would greatly benefit from programming languages specifically

designed to easily employ the features and the abstractions that JADE already provides in terms of a Java framework. This idea motivated the introduction of JADEL [5,11], a programming language based on Xtend [17], which is a dialect of Java designed to support the construction of DSLs. Although JADEL was intended to simplify the development of agents and *MAS* (*Multi-Agent Systems*), informal experiments on its use suggested that it exhibits inherent problems that severely limit its applicability. For example, programmers who were new to JADEL often preferred to directly use JADE in Java because they disconsidered Xtend as Java plus irrelevant syntactic sugar. Therefore, the main advantage of using JADEL, which is the possibility of easily adopting the abstractions that JADE provides, was not effectively perceived during the informal experiments.

Jadescript [7,13] is a fresh start toward the objectives that motivated JADEL. Jadescript is a programming language that was designed from scratch around the view of agents and MASs promoted by JADE, and it ultimately encompasses the following objectives. First, Jadescript aims at providing a clear and simple way to implement agents and related abstractions, such as ontologies [40] and behaviours [14]. Actually, the source code of a Jadescript agent bears a resemblance to the pseudocode found in textbooks on agents and MASs. Therefore, Jadescript inevitably shares several similarities with popular scripting languages, e.g., Python and JavaScript, hence its name. Second, Jadescript is intended to help programmers adopt best development practices. For example, agents should not use busy-waiting to detect events, and Jadescript natively provides cyclic behaviours [27] to allow agents to suspend when no events can be processed by available event handlers [13]. Third, Jadescript is meant to enhance the overall quality of produced software, and therefore it offers a specific type system designed to natively support the features and the abstractions that characterize JADE. Actually, the Jadescript type system [30] aims at providing very high-level abstractions for effective agent programming and at promoting the development of robust and maintainable MASs. Finally, Jadescript is intended to support mainstream development, and therefore it comes with a comprehensive set of programmer-friendly development tools. In particular, a dedicated plugin for Eclipse is proposed as the official tool to effectively use Jadescript in production environments. The Jadescript plugin [29] for Eclipse provides an integrated compiler and a set of support tools that include, e.g., dedicated graphical interfaces to manage project files and launch agents and platforms. Actually, the Jadescript plugin is designed to enhance the overall development experience of Jadescript programmers by offering a streamlined and programmer-friendly toolset to help them effectively create and manage complex projects that use Jadescript in some, or even all, parts.

This paper is organized as follows. Section 2 briefly introduces Jadescript by describing its main features and abstractions. Section 3 presents a practical use of Jadescript in a classic scenario in which agents participate in English auctions. Section 4 succinctly compares the features and the abstractions that Jadescript provides with related aspects of other programming languages. Finally, Section 5 concludes this paper by outlining some directions for future developments.

2 A Short Introduction to Jadescript

The major agent-oriented abstractions advocated by Jadescript are: agents, (agent) behaviours, and (communication) ontologies. The programmers that use JADE are well-acquainted with these names, and the abstractions that Jadescript provides purposely share similarities with their JADE counterparts. Note that, despite being designed for the implementation of MASs based on JADE, and despite being compiled to Java, Jadescript is not an object-oriented programming language. It does not provide ways to declare classes of objects, to construct objects, to invoke the methods of an object, or to access the state of an object. Actually, needed abstract data types can be defined in Jadescript by means of ontology concepts together with procedures and functions. Jadescript provides procedures and functions to define portions of reusable code with an associated visibility, which can be public or private. Functions are used to implement operations that compute a value while procedures are limited to the execution of commands. These and other features of the language are designed to direct programmers to reason about agents, their tasks, and their interactions, rather than concentrating on lower level aspects of the computation, like the organization of data in memory or the management of computational resources.

Agents represent the most relevant abstraction that Jadescript provides, and Jadescript agents inherit several characteristics from JADE agents. Since multiple agents in a MAS can share the same source code, a Jadescript agent declaration actually defines a family of agents [5] whose members have similar behaviours and share the structure of their internal states. The structure of the internal states of the agents in a given family is defined by a set of property declarations. Actually, a property is a named and statically typed part of the internal state of an agent, and it is always private to ensure that agents cannot directly access the internal states of other agents.

Jadescript supports the definition of behaviours to model the conducts of agents in terms of stateful and concurrent tasks. The behaviours that are active in an agent concurrently contribute to implement the conduct of the agent, and therefore they share the internal state of the agent. However, no race conditions on the internal state of the agent can occur because active behaviours in the same agent are executed one at a time using an internal non-preemptive scheduling mechanism. Similarly to agents, behaviour declarations can include property, function, and procedure declarations. Note that a behaviour can be bound to an agent family to limit its usage only to the agents of the specified family. This possibility has the advantage of making the private properties, functions, and procedures of an agent in the family freely accessible from the behaviour.

Currently, Jadescript behaviours are split in two categories: one-shot behaviours and cyclic behaviours. A one-shot behaviour is automatically deactivated at the end of its executions. Therefore, it is executed only once after its activations, and it represents a good way to implement atomic actions, e.g., the broadcasting of a start message to all agents in the MAS. Instead, a cyclic behaviour is normally kept in the pool of active behaviours after its execution, and it is repeatedly executed until explicitly deactivated. Therefore, it can be

used to implement repetitive actions, e.g., continuously waiting for a start message. Note that behaviours can be explicitly activated by means of the `activate` statement, and they can be deactivated using the `deactivate` statement. The activation of behaviours can be delayed [33] to occur at a specific time or after a specified delay. Moreover, cyclic behaviours can be scheduled to be executed periodically [33]. All these scheduling capabilities are essential to let agents organize their tasks in time, e.g., to implement active monitoring tasks.

Jadescript promotes event-driven programming because agents are expected to timely react to internal and external events. The reactions to these events are implemented in Jadescript using event handlers, which can be defined in agents and in behaviours. Internal events are related to the changes in the internal states of agents and behaviours. For example, an `on destroy` handler of an agent is executed by the agent right before the agent is removed from the agent container in which it is executing. Conversely, an `on create` handler of an agent is executed by the agent as soon as the agent becomes alive in order to initialize the internal state of the agent and to activate the needed behaviours. Note that `on create` handlers can have a set of named parameters. These parameters are transparently bound to the arguments provided to the agent at construction either via the command line, when the agent is created using the command line, or via external Java code, when the agent is created using the Jadescript-Java interoperability framework [32]. Moreover, note that mentioned event handlers are also available for behaviours. Actually, a behaviour can have event handlers to react to its creation, destruction, activation, deactivation, and to its selection for execution by an agent. Finally, note that other events and associated event handlers are available for agents and behaviours to handle exceptional situations and behaviour failures [31].

Currently, external events are events associated with the reception of messages. A message is characterized by a sender agent and a nonempty set of receiver agents, and all these agents are uniquely identified by means of their *AIDs* (*Agent IDentifiers*) [3], which are texts with a specific structure directly inspired by JADE AIDs. A message is also characterized by a performative [3] and a content, which is constructed by means of the ontology used for the message. Jadescript advocates an approach to communication based on asynchronous message passing, and the exchange of messages is implemented in Jadescript using asynchronous `send message` statements and message handlers. Message handlers support pattern matching [28], which allows programmers to easily express the structure of the messages that a message handler can manage. The use of pattern matching allows unifying the parts of the received message with the free variables declared in the header of the message handler, thus providing a concise and effective way to deconstruct the received message while making the relevant parts of the message explicit and readily usable.

It is common opinion that the construction of MASs can benefit from the adoption of ontologies to formalize the target application domain and to ensure that agents have a common understanding of the messages that mention the elements of the domain. Ontologies are provided in Jadescript as one of the

main abstractions of the language, and they play a central role not only in supporting communication but also in structuring the data that agents manipulate. Actually, ontologies can be associated with agents to allow agents to create and manipulate the concepts, actions, (atomic) propositions, and predicates defined in the ontology. In addition, all agents associated with the same ontology can freely exchange messages whose contents are defined using the elements of the ontology, sharing their definitions, and consequently, their meanings.

Concepts and actions are elements of ontologies used to manipulate domain-specific structured data and agent actions, respectively. They are characterized by properties, and Jadescript provides for inheritance of both concepts and actions to allow defining hierarchies of data types. Predicates and propositions are other elements of ontologies, and they are used to express facts. A predicate is associated with a lists of named and typed arguments while a proposition is not structured. Note that predicates and propositions share the `Proposition` supertype, even if programmers cannot use inheritance on these types. Finally, note that predicates and propositions are used in logical expressions, and they are also used to denote the reasons for behavior failures and exceptional situations [31].

3 English Auctions in Jadescript

This section provides a description of how Jadescript can be used to implement a MAS in which agents participate in English auctions. This example is used to show several characteristics of the language and only a few marginal details of the reported source codes were intentionally omitted. Note that this example relaxes several assumptions of ordinary toy problems, and it can be considered as genuinely more complex than the didactic examples that previous papers include to present and discuss specific features of the language.

3.1 The scenario

In the considered scenario, an agent designated as auctioneer is first created. The auctioneer knows the item it is prompted to sell, which is normally a painting, and it also knows the opening bid for the item and the reserve price. Once created, the auctioneer waits for participants to register to the auction. When at least two participants have registered, the auction starts and the auctioneer issues an initial call for bids to all registered participants. The initial call for bids includes the description of the item together with the necessary details needed to submit valid bids, namely the opening bid, the minimum increment on bids, and the deadline for submitting bids. Note that the auctioneer considers a bid as valid only if it is submitted before the deadline and if it is higher or equal to the standing bid plus the minimum increment publicized in the last call for bids. After each successful reception of a valid bid, the auctioneer issues a new call for bids to all registered participants. The new call for bids includes the current standing bid, the updated deadline, and the name of the participant who submitted the standing bid. The auctioneer continues to send updated calls for bids

until the deadline has passed and no pending bids are left. When this occurs, if the standing bid is lower than the reserve price, then the auctioneer concludes the auction without selling the item. Otherwise, the auctioneer informs all registered participants that the item is assigned to the participant that submitted the current standing bid. Note that, during the auction, participants can freely join and leave the auction. The auctioneer replies to late registrations with the latest call for bids to allow new participants to make their bids before the deadline.

3.2 The ontology

The `EnglishAuction` ontology shown in Fig. 1 is used to describe the content of each message exchanged in the MAS. The first two elements of the ontology, namely the `Participating` and the `Leaving` propositions, are used by participants to join and leave the auction, respectively.

The `Item` concept is included in the ontology to describe an item being traded in an auction. For the sake of simplicity, generic items are described using only their names. However, the scenario assumes that the auctioneer sells paintings, and therefore the `Painting` concept is included in the ontology as a a specialization of the `Item` concept. The `Painting` concept includes the `title` and the `author` properties, and when a new description of a painting is created, its name is constructed from to the title and the author of the painting.

The `SubmitBid` action is included in the ontology to be used as content for the calls for bids sent by the auctioneer to the participants. The `SubmitBid` action has several properties that specify the details of valid bids. The first property is `item`, and it is the item being traded. The second property is `currentBid`, which is either the opening bid or the standing bid. The third property is `bidMinimumIncrement`, which is the specified minimum increment. The fourth property is `deadline`, and it indicates the time at which the auctioneer will stop accepting new bids. Finally, the fifth property is `currentlyWinning`, which is the name of the participant that submitted the standing bid. If this value is the empty string, then no valid bids have been submitted yet.

The `Buy` action is included in the ontology to denote the act of buying the specified item, while the `Priced` predicate is used to associate an item with a price. The `Buy` and the `Priced` elements are both used as content of messages sent by participants to submit new bids, while the `BidRejected` predicate is used by the auctioneer to refuse a bid indicating the reason that caused the bid to be rejected. This reason is described using one of the following predicates of the ontology. The `BidTooLow` predicate indicates that the submitted bid was too low, and it includes a property that specifies the minimum value for a valid bid. The `InvalidBid` predicate is used to generically reject a bid by providing a textual explanation of what went wrong during the submission of the bid.

Finally, two predicates of the ontology are used by the auctioneer to inform registered participants of the outcome of the auction. The `ItemNotSold` predicate includes a property to specify the item the auctioneer failed to sell. The `ItemSold` predicate includes two properties, namely `buyer`, which contains the AID of the winning participant, and `price`, which contains the price of the winning bid.

```
ontology EnglishAuction
  proposition Participating
  proposition Leaving
  concept Item(name as text)
  concept Painting(author as text, title as text)
    extends Item with name = title+" by "+author
  action SubmitBid(item as Item, currentBid as integer,
    bidMinimumIncrement as integer,
    deadline as timestamp, currentlyWinning as text)
  action Buy(item as Item)
  predicate Priced(item as Item, price as integer)
  predicate BidRejected(reason as Proposition)
  predicate BidTooLow(minimumBid as integer)
  predicate InvalidBid(otherReason as text)
  predicate ItemSold(item as Item, buyer as aid,
    finalPrice as integer)
  predicate ItemNotSold(item as Item)
```

Fig. 1. The Jadescript implementation of the English auction ontology.

3.3 The auctioneer

The Jadescript source code for the auctioneer is shown in Fig. 2. The properties defined in the declaration of the auctioneer constitute the internal state of the auctioneer, which equals the state of the auction for the sake of simplicity. These properties can be subdivided in three groups. The first group contains the predefined parameters of the auctioneer. In particular, the `minimumParticipants` property specifies the minimum number of participants required for an auction to start. The auctioneer waits until the number of participants that registered to the action reaches the specified minimum number. The amount of time that the auctioneer waits for new bids after sending a call for bids is denoted by the `waitingForBidsTime` property. The `startBid` property specifies the opening bid, and the `bidMinimumIncrement` property denotes the minimum required increment between two subsequent bids. The `reserve` property denotes the reserve price. Finally, the last property of this group is the `item` property, which denotes the item being traded.

The second group of properties that constitute the internal state of the auctioneer is used to track the dynamic state of the current auction. The `currentBid` property denotes the standing bid, and it is initialized with the mentioned `startBid` property. The `candidateBuyer` property is used to store the identity of the participant that submitted the standing bid when at least one valid bid has been received, which is an event that is denoted by the `thereIsCandidate` property. Finally, the `participants` property is a set of AIDs used to store the identities of all registered participants. This set is updated dynamically by the auctioneer every time a participant registers or deregisters.

```
agent Auctioneer uses ontology EnglishAuction
  property minimumParticipants = 2
  property waitingForBidsTime = "PT30S" as duration
  property startBid = 80
  property reserve = 120
  property bidMinimumIncrement = 2
  property item = Painting("Leonardo", "Mona Lisa")

  property currentBid = startBid
  property candidateBuyer as aid
  property thereIsCandidate = false
  property participants as set of aid

  property doAuction = DoAuction
  property endAuction = EndAuction

  on create do
    log "Agent "+name of agent+" created."
    activate AwaitParticipants
```

Fig. 2. The Jadescript implementation of the auctioneer.

The last group of properties that constitute the internal state of the auctioneer contains two properties that refer to behaviours. The first property refers to a `DoAuction` behaviour, and the auctioneer uses this behaviour to run the auction. The second property refers to an `EndAuction` behaviour, and the auctioneer uses this behaviour to finalize the auction by informing all registered participants of the outcome of the auction. Note that these two properties refer to behaviours that are explicitly activated and deactivated when needed.

The agent declaration shown in Fig. 2 is concluded with an `on create` handler. As soon as the auctioneer starts, it writes a message to its log, and then it activates an `AwaitParticipants` behaviour, whose source code is shown in Fig. 3. The activated `AwaitParticipants` behaviour performs the task of waiting for a sufficient number of participants to register to the auction. This behaviour is designed to be used exclusively by the auctioneer, and therefore its declaration uses the `for agent` clause in its header. This tight link between the `AwaitParticipants` behaviour and the auctioneer has two relevant consequences. First, the behaviour can refer to the properties of the agent, which are always private. In particular, the `minimumParticipants` and the `participants` properties are used by the behaviour. Second, the behaviour is transparently associated with the ontologies used by the agent. In this case, this is used to access to the `Participating` and the `Leaving` propositions.

The `AwaitParticipant` behaviour created by the auctioneer to manage the start of the auction uses the two message handlers shown in Fig. 3. The first handler processes inform messages that contain a `Participating` proposition.

```
cyclic behaviour AwaitParticipants for agent Auctioneer
  on message inform Participating do
    add sender of message to participants
    if size of participants >= minimumParticipants do
      log "Starting auction."
      log "Selling: "+item+"."
      activate doAuction
      deactivate this

  on message inform Leaving do
    remove sender from participants

  on activate do
    do log "Waiting for participants."
```

Fig. 3. The Jadescript implementation of the behaviour used to wait for participants.

These messages are sent by agents interested in participating in the auction, and therefore their AIDs are added to the set of participants. If, after this addition, the number of registered participants reaches the specified minimum number of participants, then the auctioneer changes its behaviour by deactivating its current behaviour and by activating the `doAuction` behaviour. The second handler shown in Fig. 3 processes inform messages that contain a `Leaving` proposition. These messages are sent by participants that want to leave the auction, and therefore their AIDs are removed from the set of participants. Finally, note that the `AwaitParticipant` behaviour created by the auctioneer writes a message to its log when activated.

The task of running an auction is implemented by the auctioneer using a `DoAuction` behaviour, whose declaration is shown in Fig. 4. This behaviour assumes that a sufficient number of participants is registered to the auction. As soon as the behaviour is activated, the actioneer executes the `callForBids` procedure. This procedure sends a call for proposals to all registered participants. This call for proposals contains a `SubmitBid` action that details the information needed by participants to submit valid bids. After sending the call for proposals, the `DoAuction` behaviour performs a delayed activation of the behaviour used to terminate the auction, which is referenced by the `endAuction` property. The time at which this behaviour will be activated is stored in the `nextTimeout` property, which is computed as `now + waitingForBidsTime`. Note that each call to the `callForBids` procedure resets this delayed activation, thus postponing the activation of the behaviour used to terminate the auction.

The behaviour used by the auctioneer to run auctions also handles the reception of inform messages that mention either the `Participating` or the `Leaving` propositions in order to dynamically manage the set of registered participants. In particular, when an inform message from an agent who wants to join the auction arrives, the auctioneer adds the agent to the set of participants, and it

```
cyclic behaviour DoAuction for agent Auctioneer
  property nextTimeout as timestamp

  on activate do
    do callForBids

  procedure callForBids do
    nextTimeout = now + waitingForBidsTime
    do sendCFPMessage with bidders = participants
    activate endAuction at nextTimeout

  procedure sendCFPMessage with bidders as set of aid do
    currentlyWinning = ""
    if thereIsCandidate do
      currentlyWinning = name of candidateBuyer
    send message cfp SubmitBid(item, currentBid,
      bidMinimumIncrement, nextTimeout,
      currentlyWinning) to bidders

  on message inform Participating do
    add sender of message to participants
    do sendCFPMessage with bidders = { sender }

  on message inform Leaving do
    remove sender of message from participants
    if size of participants < 2 do
      activate endAuction

  on message propose (Buy(proposedItem),
    Priced(proposedItem, proposedPrice)) do
    minBid = currentBid + bidMinimumIncrement
    if proposedPrice < minBid do
      send message reject_proposal (Buy(proposedItem),
        Priced(proposedItem, proposedPrice),
        BidRejected(BidTooLow(minBid)))
        to sender of message
    else do
      send message accept_proposal (
        Buy(proposedItem),
        Priced(proposedItem, proposedPrice)
      ) to sender of message
      currentBid = proposedPrice
      thereIsCandidate = true
      candidateBuyer = sender of message
      do callForBids
```

Fig. 4. The Jadescript implementation of the behaviour used to run auctions.

replies to the agent with a call for proposals. This call for proposals is populated with the needed information for the new participant to place valid bids before the deadline. Similarly, when an inform message from a participant that wants to leave the auction arrives, the auctioneer removes the AID of the sender from the set of participants. Note that if the number of registered participants is less than two, the auctioneer immediately terminates the auction.

Finally, note that the `DoAuction` behaviour used by the auctioneer to run the auction provides a message handler for proposals, as shown in the bottom of Fig. 4. In order for this handler to be executed, the content of the message must match against the pattern composed of a pair of types `(Buy, Priced)`. If the received message successfully matches against this pattern, the message handler is executed and it can access the `proposedItem` and the `proposedPrice` variables. The values of these variables are transparently extracted from the content of the message during the matching against the specified pattern. Therefore, these values can be freely used to verify the validity of the received proposal. First, the auctioneer ensures that the bid is valid by checking that the proposed price is sufficiently high. In particular, the proposed price must be higher than or equal to `currentBid + bidMinimumIncrement`. If the proposed price is not sufficiently high, the bid is rejected with an appropriate reason for the rejection. Otherwise, the auctioneer accepts the bid, and the state of the auction is updated to take into account the new standing bid. In particular, a new iteration of the auction is immediately started by calling the `callForBids` procedure. Note that if no valid bids are submitted by the deadline, the delayed activation of the behaviour used to terminate the auction ensures that the auction is still terminated.

The source code of the behaviour used to terminate an auction is shown in Fig. 5. The auctioneer uses a delayed activation of this behaviour to ensure that the auction terminates at the appropriate deadline. When activated, this behaviour first deactivates the behaviour used to run the auction, which is referenced by the `doAuction` property, in order to prevent it from accepting bids submitted after the deadline. Then, it checks the final state of the auction to compute its outcome. If there is no valid standing bid higher than the reserve price, the auction is concluded with no transactions. In this case, the auctioneer informs all participants that the item was not sold. On the contrary, if a valid standing bid is available, the auctioneer notifies all participants of the successful outcome, and it indicates the identity of the winner of the auction. Finally, note that some corner cases were intentionally omitted for the sake of simplicity. For example, the auctioneer does not treat sufficiently well the case of a participant that leaves the auction while it is the current winner.

3.4 The participants

Together with the auctioneer, the MAS comprises a set of participants. Even if the Jadescript source code of participants, in the `Bidder` agent declaration, is not shown for space constraints, participants are very simple and they can be easily described. First, participants use the `EnglishAuction` ontology to share

```
one shot behaviour EndAuction for agent Auctioneer
  on activate do
    deactivate doAuction
    if not thereIsCandidate or currentBid < reserve do
      log "No valid bid submitted. Not selling the item."
      send message inform ItemNotSold(item)
        to participants
    else do
      log "Selling item "+item+" to "+candidateBuyer+"."
      send message inform ItemSold(item, candidateBuyer,
        currentBid) to participants
```

Fig. 5. The Jadescript implementation of the behaviour used to terminate auctions.

the definition of concepts, actions, predicates, and propositions with the auctioneer. Then, each participant has a `budget` property that stores the amount of money available for the auction. Once created, participants immediately activate a `ParticipateToAuction` behaviour to enter the auction and try to win it. The implemented strategy adopted by participants to try to win the auction is very simple: a participant always proposes the minimum price sufficient to make the proposal valid, and it stops bidding only if it does not have enough money to make a valid proposal.

The source code of the `ParticipateToAuction` behaviour used by participants to participate to the auction is shown in Fig. 6. This behaviour is defined by several event handlers. Upon activation and deactivation of this behaviour, the participant informs the auctioneer about its interest to participate to the auction. When a call for proposals arrives, the corresponding message handler deconstructs it and uses its parts to compute the decision on what to do. In particular, if the participant is not the one that submitted the current standing bid, and if the participant has enough money and time to propose a higher bid, then the participant proposes a higher bid by sending the corresponding message to the auctioneer. The proposal is then either accepted or rejected by the auctioneer. These two events are handled in the participant by the two message handlers that match against accept proposals and reject messages. Note that the behaviour also handles the final outcome of the auction, providing a message handler for each one of the possible messages sent by the auctioneer to inform participants of the termination of the auction.

4 Related Work

Several *AOP* (*Agent-Oriented Programming*) languages have been developed in the last few decades to provide effective tools to support a novel programming paradigm [39] suitable to develop agents and MASs. Besides languages mostly intended for theoretical purposes, like AGENT0 [38] and AgentSpeak(L) [36], notable examples of AOP languages intended for practical applications are Jason,

```
cyclic behaviour ParticipateToAuction for agent Bidder
  on activate do
    send message inform Participating to aid("Auctioneer")

  on deactivate do
    send message inform Leaving to aid("Auctioneer")

  on message cfp (SubmitBid(item, currentBid,
    bidMinimumIncrement, deadline, currentWinner), _) do
    bid = currentBid + bidMinimumIncrement
    if currentWinner != name of agent
      and now < deadline and bid <= budget do
      log "Submitting bid: "+bid+"."
      activate SendPropose(item, bid)
    else if bid > budget do
      log "Not enough money, giving up."

  on message accept_proposal do
    log "My bid has been accepted."

  on message reject_proposal (_, _, reason) do
    log "My bid was rejected, reason: "+reason

  on message inform ItemSold(item, aid of agent, bid) do
    log "I bought "+item+" for "+bid+"!"

  on message inform ItemSold(item, other, bid) do
    log other+" bought "+item+" for "+bid+"."

  on message inform ItemNotSold(item) do
      log "Item not sold: "+item+"."
```

Fig. 6. The Jadescript implementation of the behaviour used by participants.

ASTRA, and SARL. In the remaining of this section, the main features and abstractions of these three languages are outlined and compared with the related features and abstractions that Jadescript advocates.

AgentSpeak(L) [36] is a well-known AOP language that was formalized to provide an operational proof-theoretic semantics to reason on *BDI* (*Belief-Desire-Intention*) agents. In AgentSpeak(L), agent programs are expressed as logic programs, and they are composed of beliefs, goals, and plans. Jason [18], which is one of the most popular implementations of AgentSpeak(L), has gained significant popularity in recent years. Jason extends AgentSpeak(L) with several features, like a specific support for interoperability with Java. The tight link between Jason and Java is so relevant that Jason agents are expected to be situated in environments implemented in Java, and several parts of the Jason

interpreter can be customized by extending the core Java classes of the interpreter. ASTRA [19,20] is another implementation of AgentSpeak(L), and it also provides specific extensions. ASTRA extends AgentSpeak(L) by introducing several features inspired by the literature on agents and MAS, e.g., a support for teleo-reactive [26] functions with encapsulated rules.

Jadescript and the mentioned implementations of AgentSpeak(L) are all AOP languages intended for practical uses. However, Jadescript has some significant differences with respect to AgentSpeak(L) and its derivates, one of the most significant of which is in the approach to programming agents. AgentSpeak(L) is a language that uses the BDI model to program agents, while Jadescript, on the other hand, is both an imperative and an event-driven programming language. Actually, while the focus in AgentSpeak(L) is on describing the mental attitudes of the agents, the focus in Jadescript is on specifying the tasks performed by the agents and on structuring the interactions among agents in the MAS. Another key difference between Jadescript and the languages that derive from AgentSpeak(L) is in the syntax and the semantics of the language. AgentSpeak(L) is based on logic programming, and it uses a syntax that is similar to Prolog. Jadescript, on the other hand, has a syntax that is closer to modern scripting languages like Python and JavaScript. Therefore, Jadescript is more accessible to mainstream programmers, who are not supposed to be familiar with logic programming and with the declarative paradigm. Finally, it is worth noting that Jadescript is specifically designed to use JADE, whereas the mentioned practical implementations of AgentSpeak(L) can be used with a variety of agent platforms. Therefore, Jadescript is a better choice for developers who are already familiar with JADE, also because its main features and abstractions are inspired by the corresponding features and abstractions provided by JADE.

Differently from AgentSpeak(L) and its implementation, SARL [37] is an AOP language that can be considered as imperative and event-driven. SARL is equipped with a syntax that is easy to understand for users of mainstream programming languages. One of its most noteworthy features is its support for holonic agents, which are agents composed of other agents. Moreover, SARL is designed to not be tied to any particular platform, although it is frequently used with Janus [37]. SARL has several similarities with JADEL, which is the predecessor of Jadescript. For example, SARL and JADEL have similar syntaxes to define agents, and they use similar linguistic constructs to handle events. Moreover, both SARL and JADEL include specific extensions of Xtend for the imperative parts of the source codes of agents. Despite these similarities, SARL and JADEL were developed independently and have distinct purposes.

The main difference between SARL and Jadescript is that SARL explicitly supports object-oriented programming, while Jadescript is a pure AOP language. Actually, SARL supports the definition of classes and the manipulation of objects alongside the declaration of agents and of their tasks. On the other hand, Jadescript purposely excludes the concepts of object-oriented programming from the language to offer agent-oriented abstractions as valid alternatives to promote reusability and composability [10].

5 Conclusion

Jadescript is a promising tool to develop real-world MASs that target mission-critical applications and services (e.g., [6,25]). Its unique combination of simplicity and conciseness makes it a valuable addition to the toolkit of the programmers of agents and MASs. However, Jadescript is still in its early stages, with early versions of the compiler and associated tools having only recently been made available to the open source community (github.com/aiagents/jadescript). Therefore, Jadescript presents significant opportunities for further developments.

One promising direction for extending Jadescript is to incorporate *IPs* (*Interaction Protocols*) [34] as a primary abstraction of the language. IPs are intended to precisely specify the possible patterns of the interactions among agents, and therefore their support in Jadescript requires linguistic constructs to allow specifying new IPs and to allow agents to enact IPs on the basis of these specifications. Actually, by defining the role of an agent within an IP, the designer of the agent is guided to design the behaviours of the agent taking into account the expected interactions of the agent in the scope of the IP. This approach to the design of agents and behaviours has the beneficial effect of improving the clarity and the modularity of the design, and it also eases the identification of reusable agents and behaviours for common communication patterns.

Finally, another possible development of Jadescript is about providing effective language-level features to enable the use of epistemic and intentional propositions in the agents. This extension is expected to provide Jadescript with a more expressive way to describe the decision making processes and cognitive abilities of the agents, thus ultimately improving the robustness, the maintainability, and the reusability of agents and MASs.

Acknowledgements This work was partially supported by the Italian Ministry of University and Research under the PRIN 2020 grant 2020TL3X8X for the project *Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems* (T-LADIES).

References

1. Adorni, G., Bergenti, F., Poggi, A., Rimassa, G.: Enabling FIPA agents on small devices. In: Proceedings of the 5th International Workshop on Cooperative Information Agents (CIA 2001). Lecture Notes in Artificial Intelligence, vol. 2182, pp. 248–257. Springer (2001)
2. Bellifemine, F., Bergenti, F., Caire, G., Poggi, A.: JADE–A Java Agent DEvelopment Framework. In: Multi-Agent Programming, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 25, pp. 125–147. Springer (2005)
3. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley Series in Agent Technology, John Wiley & Sons (2007)
4. Bergenti, F.: A discussion of two major benefits of using agents in software development. In: Proceedings of the 3rd International Workshop on Engineering Societies in the Agents World (ESAW 2002). Lecture Notes in Artificial Intelligence, vol. 2577, pp. 1–12. Springer (2003)

5. Bergenti, F.: An introduction to the JADEL programming language. In: Proceedings of the 26$^{\text{th}}$ IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2014). pp. 974–978. IEEE (2014)
6. 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)
7. Bergenti, F., Caire, G., Monica, S., Poggi, A.: The first twenty years of agent-based software development with JADE. Autonomous Agents and Multi-Agent Systems **34**(36) (2020)
8. Bergenti, F., Franchi, E., Poggi, A.: Agent-based social networks for enterprise collaboration. In: Proceedings of the 20$^{\text{th}}$ IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2011). pp. 25–28. IEEE (2011)
9. Bergenti, F., Gleizes, M.P., Zambonelli, F. (eds.): Methodologies and Software Engineering for Agent Systems. Springer (2004)
10. Bergenti, F., Huhns, M.N.: On the use of agents as components of software systems. In: Methodologies and Software Engineering for Agent Systems. pp. 19–31. Springer (2004)
11. Bergenti, F., Iotti, E., Monica, S., Poggi, A.: Agent-oriented model-driven development for JADE with the JADEL programming language. Computer Languages, Systems & Structures **50**, 142–158 (2017)
12. Bergenti, F., Monica, S.: Location-aware social gaming with AMUSE. In: Proceedings of the 14$^{\text{th}}$ International Conference on Advances in Practical Applications of Scalable Multi-agent Systems (PAAMS 2016). Lecture Notes in Computer Science, vol. 9662, pp. 36–47. Springer (2016)
13. Bergenti, F., Monica, S., Petrosino, G.: A scripting language for practical agent-oriented programming. In: Proceedings of the 8$^{\text{th}}$ 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 (2018)
14. Bergenti, F., Petrosino, G.: Overview of a scripting language for JADE-based multi-agent systems. In: Proceedings of the 19$^{\text{th}}$ Workshop "From Objects to Agents" (WOA 2018). CEUR Workshop Proceedings, vol. 2215, pp. 57–62. RWTH Aachen (2018)
15. Bergenti, F., Poggi, A.: Ubiquitous information agents. International Journal of Cooperative Information SystemsVolume **11**(3–4), 231–244 (2002)
16. Bergenti, F., Poggi, A.: Developing smart emergency applications with multi-agent systems. International Journal of E-Health and Medical Communications **1**(4), 1–13 (2010)
17. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing (2013)
18. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley Series in Agent Technology, John Wiley & Sons (2007)
19. Collier, R.W., Russell, S., Lillis, D.: Reflecting on agent programming with AgentSpeak(L). In: Proceedings of 4$^{\text{th}}$ the International Conference of Principles and Practice of Multi-Agent Systems (PRIMA 2015). Lecture Notes in Computer Science, vol. 9387, pp. 351–366. Springer (2015)

20. Dhaon, A., Collier, R.: Multiple inheritance in AgentSpeak(L)-style programming languages. In: Proceedings of the 4th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2014) at ACM SIGPLAN Conference Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2014). pp. 109–120. ACM (2014)
21. Federico, B., Agostino, P.: Agent-based approach to manage negotiation protocols in flexible CSCW systems. In: Proceedings of the 4th International Conference on Autonomous Agents (AGENTS 2000). pp. 267–268. ACM (2000)
22. Fowler, M., Parsons, R.: Domain Specific Languages. Addison-Wesley Signature, Addison-Wesley (2010)
23. Iotti, E., Petrosino, G., Monica, S., Bergenti, F.: Exploratory experiments on programming autonomous robots in Jadescript. In: Proceedings of the 1st Workshop on Agents and Robots for Reliable Engineered Autonomy (AREA 2020) at the European Conference on Artificial Intelligence (ECAI 2020). Electronic Proceedings in Theoretical Computer Science, vol. 319. Open Publishing Association (2020)
24. Iotti, E., Petrosino, G., Monica, S., Bergenti, F.: Two agent-oriented programming approaches checked against a coordination problem. In: Proceedings of the 17th International Conference on Distributed Computing and Artificial Intelligence (DCAI 2020). pp. 60–70. Springer (2021)
25. Monica, S., Bergenti, F.: A comparison of accurate indoor localization of static targets via WiFi and UWB ranging. In: Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection. pp. 111–123. Springer (2016)
26. Nilsson, N.J.: Teleo-reactive programs for agent control. Journal of Artificial Intelligence Research **1** (1993)
27. Petrosino, G., Bergenti, F.: An introduction to the major features of a scripting language for JADE agents. In: Proceedings of the 17th Conference of the Italian Association for Artificial Intelligence (AI*IA 2018). Lecture Notes in Artificial Intelligence, vol. 11298, pp. 3–14. Springer (2018)
28. 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. RWTH Aachen (2019)
29. Petrosino, G., Iotti, E., Monica, S., Bergenti, F.: Prototypes of productivity tools for the Jadescript programming language. In: Proceedings of the 22nd Workshop "From Objects to Agents" (WOA 2021). CEUR Workshop Proceedings, vol. 2963, pp. 14–28. RWTH Aachen (2021)
30. Petrosino, G., Iotti, E., Monica, S., Bergenti, F.: A description of the Jadescript type system. In: Proceedings of the 3rd International Conference on Distributed Artificial Intelligence (DAI 2022). Lecture Notes in Computer Science, vol. 13170, pp. 206–220. Springer (2022)
31. Petrosino, G., Monica, S., Bergenti, F.: Robust software agents with the Jadescript programming language. In: Proceedings of the 23rd Workshop "From Objects to Agents" (WOA 2022). CEUR Workshop Proceedings, vol. 3261, pp. 194–208. RWTH Aachen (2022)
32. Petrosino, G., Monica, S., Bergenti, F.: Cross-paradigm interoperability between Jadescript and Java. In: Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023). vol. 1, pp. 165–172. Science and Technology Publications (2023)

33. Petrosino, G., Monica, S., Bergenti, F.: Delayed and periodic execution of tasks in the Jadescript programming language. In: Proceedings of the 19$^{\text{th}}$ International Conference on Distributed Computing and Artificial Intelligence (DCAI 2022). pp. 50–59. Springer (2023)
34. Poslad, S.: Specifying protocols for multi-agent systems interaction. ACM Transactions on Autonomous and Adaptive Systems **2**(4), 15:–15:24 (2007)
35. Poslad, S., Charlton, P.: Standardizing agent interoperability: The FIPA approach. In: Multi-Agent Systems and Applications. Lecture Notes in Computer Science, vol. 2086. Springer (2001)
36. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: MAAMAW 1996: Agents Breaking Away. pp. 42–55. Springer (1996)
37. Rodriguez, S., Gaud, N., Galland, S.: SARL: A general-purpose agent-oriented programming language. In: Proceedings of the IEEE/WIC/ACM International Joint Conferences of Web Intelligence (WI 2014) and Intelligent Agent Technologies (IAT 2014). vol. 3, pp. 103–110. IEEE (2014)
38. Shoham, Y.: AGENT0: A simple agent language and its interpreter. In: Proceedings of the 9$^{\text{th}}$ National Conference on Artificial Intelligence (AAAI 1991). vol. 91, pp. 704–709 (1991)
39. Shoham, Y.: Agent-oriented programming. Artificial Intelligence **60**(1), 51–92 (1993)
40. Tomaiuolo, M., Turci, P., Bergenti, F., Poggi, A.: An ontology support for semantic aware agents. In: Proceedings of the 7$^{\text{th}}$ International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS 2005). Lecture Notes in Computer Science, vol. 3529, pp. 140–153. Springer (2006)