Article

Flexible Agent Architecture: Mixing Reactive and Deliberative Behaviors in SPADE

Javier Palanca *,†, Jaime Andres Rincon †, Carlos Carrascosa †, Vicente Javier Julian † and Andrés Terrasa †

Valencian Research Institute for Artificial Intelligence (VRAIN), Universitat Politècnica de València, Camí de Vera s/n, 46022 Valencia, Spain
* Correspondence: jpalanca@dsic.upv.es
† These authors contributed equally to this work.

Abstract: Over the years, multi-agent systems (MAS) technologies have shown their usefulness in creating distributed applications focused on autonomous intelligent processes. For this purpose, many frameworks for supporting multi-agent systems have been developed, normally oriented towards a particular type of agent architecture (e.g., reactive or deliberative agents). It is common, for example, for a multi-agent platform supporting the BDI (Belief, Desire, Intention) model to provide this agent model exclusively. In most of the existing agent platforms, it is possible to develop either behavior-based agents or deliberative agents based on the BDI cycle, but not both. In this sense, there is a clear lack of flexibility when agents need to perform part of their decision-making process according to the BDI paradigm and, in parallel, require some other behaviors that do not need such a deliberation process. In this context, this paper proposes the introduction of an agent architecture called Flexible Agent Architecture (FAA) that supports the development of multi-agent systems, where each agent can define its actions in terms of different computational models (BDI, procedural, neural networks, etc.) as behaviors, and combine these behaviors as necessary in order to achieve its goals. The FAA architecture has been integrated into a real agent platform, SPADE, thus extending its original capabilities in order to develop applications featuring reactive, deliberative, and hybrid agents. The integration has also adapted the existing facilities of SPADE to all types of behaviors inside agents, for example, the coordination of agents by using a presence notification mechanism, which is a unique feature of SPADE. The resulting SPADE middleware has been used to implement a case study in a simulated robotics scenario, also shown in the paper.

Keywords: BDI agents; multi-agent systems; artificial intelligence

1. Introduction

Multi-agent systems (MAS) technology has evolved from the 1990s to the present day enabling the design and development of distributed intelligent systems in a multitude of domains. Nevertheless, there has been a period of certain decline caused by the multitude and variety of existing platforms, the lack of standards, as well as the feeling of not achieving the initial high expectations. However, the emergence in recent years of new distributed areas, such as the Internet of Things (IoT) [1,2], smart grids [3,4], cyber–physical systems [5,6], autonomous cars or drones [7,8], or Smart Cities [9,10], which require the automation and interconnection of intelligent devices, has fostered a new evolution of this type of systems. Multi-agent systems technology is highly appropriate for such domains since it facilitates the emergence of collective intelligence and social behaviors. Regarding the evolution of tools and frameworks for the design and development of multi-agent systems, new approaches have continued to appear over the last few years. The review presented in [11] makes a comprehensive analysis of current platforms for developing MAS. This work concludes with the necessity of being aware of the existing approaches in order to select the framework which best fits each scientific need, as well as encouraging new theoretical and practical developments in the MAS area.

Taking this into account, and in the opinion of the authors of this paper, the area of agent architectures is one of the key aspects which still needs to improve in order to fit the requirements of the aforementioned new domains. There are many different types of agent architectures, which, in most cases, are oriented towards a particular domain or problem type where they are best suited. In this context, the work presented in this paper tries to extend the functionality of an existing behavior-based architecture by incorporating BDI (Belief, Desire, Intention) reasoning capabilities, with the main goal of proposing a hybrid architecture that may be applied to a wide variety of current domains, especially the ones mentioned above. The idea is to have agents that can flexibly separate their decision-making capacity internally, where part of it can follow a BDI cycle but can also, in parallel, make decisions that do not require such a deliberation process by using other behaviors.

On the other hand, coordination is a key aspect of developing any multi-agent system. In many cases, its design is complex since it implies that different computational entities know at all times what the other entities are doing in order to adapt their tasks. In this case, existing solutions do not facilitate its implementation and that is why it is necessary to have tools that facilitate coordination that is simple and as less explicit as possible between autonomous agents.

According to these aspects, this paper introduces the Flexible Agent Architecture (FAA), which proposes the integration of behaviors of different types, specifically BDI behaviors, in the same agent. This architecture improves the options in which decisions can be made internally. In addition, mechanisms are incorporated that facilitate in a transparent way the sharing of knowledge between the BDI behavior and the rest of the agent's behaviors. Moreover, the FAA incorporates capabilities to know the status of other agents in order to facilitate coordination by using a presence mechanism similar to that used in a chat application.

In this way, a multi-agent system, that incorporates this flexible agent architecture, would be able to include pure reactive or deliberative agents or even hybrid ones. Moreover, it will also be able to offer coordination mechanisms in a simpler and more elegant way among all the agents that compose the multi-agent system. This architecture has been implemented in the SPADE agent platform, which is a multi-agent platform that allows the creation of behavior-based, multi-agent applications. One of the main features of *SPADE* is its flexibility, which facilitates adding new features to agents as needed. As a result of the work presented in this paper, SPADE now features the possibility of integrating new types of behaviors (tasks) in the same agent incorporating a BDI reasoning cycle based on *AgentSpeak* [12] and also facilitating coordination through the use of a presence mechanism.

The paper includes a case study, which makes use of the proposed FAA, where different robotic arms must coordinate in order to move objects simulating a manufacturing process in the context of a factory. This example illustrates how the new features available for *SPADE* agents allow them to coordinate and mix middle and long-term planning tasks with direct perception and classification actions in the context of a realistic (although simulated) environment.

The rest of the paper is structured as follows. Section 2 analyses previous work in the area of BDI and its supporting frameworks. Section 3 introduces the SPADE platform for developing multi-agent systems in Python. Section 4 presents the Flexible Agent Architecture and details how it has been implemented in the SPADE platform. Section 5 presents a case study where a multi-agent system implemented in SPADE and following the FAA has been used in the context of a simulated robotics environment. Finally, Section 6 presents some conclusions of the paper.

2. State of the Art

The use of descriptive agents, programmed by following the Belief, Desire, and Intention (BDI) paradigm, has proven to be very useful in several application areas of multi-agent systems, such as ambient intelligence, IoT systems, cyber–physical systems, etc. However, the feasibility of such applications has largely to do with the existence of

appropriate frameworks which may enable their actual development. This section presents a brief review of this paradigm and a collection of frameworks in which it has been applied over the years.

The BDI paradigm for agents, formalized by Rao and Georgeff in [13], was initially founded on a psychological model by Bratman [14] about how practical reasoning works in humans. This paradigm classifies the knowledge of an agent into three different parts: the agent's internal representation of both the environment state and its own internal state (Beliefs); the goals which are available to the agent in the current situation (Desires); and the goals that the agent is committed to (Intentions). Among other benefits of the BDI paradigm when applied to multi-agent systems, one key advantage of this model is that, by imitating human-like behavior, it allows for the representation of complex reasoning in a comprehensible way. This aspect enables end-users to analyze and modify decision-making algorithms more easily than other representation and reasoning paradigms. In fact, this paradigm is similar to rule-based approaches paradigms as SWRL [15].

There are many examples of general-purpose multi-agent system frameworks and developments which have adopted the BDI paradigm or have integrated it to some extent. Chronologically speaking, the first approaches used *LISP* as their programming language, such as *AGENT0* [16]; but, since then, almost all the relevant frameworks using BDI have been based on Java technology. Some of them are extensions of existing platforms, as, for example, the *JADEX* (https://sourceforge.net/projects/jadex/files/, accessed on 7 January 2023) [17] or *BDI4JADE* (https://www.inf.ufrgs.br/prosoft/bdi4jade/, accessed on 7 January 2023) [18] extensions of the *JADE* (https://jade.tilab.com, accessed on 7 January 2023) [19] platform. *JIAC*(https://www.jiac.de, accessed on 7 January 2023) (Java-based Intelligent Agent Componentware) [20] and its lightweight version for constrained devices, *microJIAC*, are addressed to large-scale distributed applications and services based on Java and BDI with its own rule engine. In this group, there are also commercial products, such as *JACK* (https://aosgrp.com/products/jack/, accessed on 7 January 2023) [21], which has its own plan language that compiles to Java classes for execution.

Other relevant results are related to the development of agent programming languages to implement BDI agents, such as *2APL* and *AgentSpeak*, which are declarative languages based on logic programming. *2APL* [22] is an agent programming language that adds the BDI concepts to *3APL* [23], with JADE as its underlying platform. A similar approach is followed by the *Jason* (http://jason.sourceforge.net, accessed on 7 January 2023) [24] framework, which implements the *AgentSpeak* language. Apart from *Jason*, there are other *AgentSpeak* implementations, as for example *LightJason* (https://lightjason.org, accessed on 7 January 2023) [25], which is a completely new implementation of a Jason-like platform, or *agentspeak* (https://pypi.org/project/agentspeak/, accessed on 7 January 2023), implemented in Python.

In the last few years, several lines of work related to applying the BDI paradigm to specific domains have emerged. These lines can be broadly categorized into two groups: the first one is devoted to making the BDI paradigm more usable by creating frameworks that combine this paradigm with other technologies in the context of specific domains, and the second one is related to integrating rational (BDI) agents into agent-based simulation platforms. These two groups are now briefly reviewed.

A very interesting example of a BDI-based platform for a specific domain is the work presented in [26], which introduces PROFETA (Python Robotic Framework for Designing Strategies), a framework in Python for programming BDI agents using a slight modification of AgentSpeak(L). PROFETA is mainly addressed for programming robotic systems, combining object-oriented code, which is useful for robotic devices, and declarative code, which is very powerful for defining decision-making behaviors in autonomous robots. Following this line of work, there have been some recent developments as the application of BDI to control robots with a ROS2 operating system [27] or even extend the model for including real-time restrictions [28]. In another approach in this group, the authors of [29,30] present the Jason-RS architecture, which gives the REST Web Service ability to the

BDI agents running in the Jason framework. In particular, agents can exhibit functionality as a Web Rest Service using the proposed architecture, which facilitates communication with IoT servers. Moreover, in [31], authors present Janus combined with the GORITE BDI agent framework. This approach provides a methodology for developing agent-based cyber–physical systems in industrial environments. GORITE BDI is a BDI framework that uses explicit goal representations in order to overcome the limitations in this aspect of previous BDI frameworks. Following the idea of improving and enriching the BDI paradigm, Alzetta et al. [32] presents a revision of the BDI model by integrating real-time mechanisms into the reasoning cycle of the agent. The main idea of this approach is to adapt the reasoning cycle to the needs of distributed cyber–physical systems which are based on real-time embedded systems, in order to guarantee some time constraints. Finally, BDI has also been the basis for interesting developments centered on the mental state of agents, including emotions and personality, as, for example, *EBDI* [33] and *ABC-EBDI* [34]. Moreover, recently there have also been new additions in the BDI model, as the modularity concept [35] which allows for grouping not only BDI concepts at the agent level but also at the environment level.

The second group, which comprises developments related to introducing BDI models into agent-based simulations, also exhibits several recent results. In [36], authors study the advantages of using agent programming languages and logic, such as BDI-based languages, for agent-based simulation. Authors detect three alternatives: implementing simulation features over an agent programming platform, implementing a BDI model in an agent-based simulation platform, or, the best option, combining agent-based simulation platforms with agent programming platforms as proposed in [37], where authors integrate the Jason agent programming language in the AORTA framework [38]. Another interesting study is presented in [39], where an in-depth analysis of the integration of BDI agents in simulation platforms and a general integration framework are proposed. Following a different approach, in [40], authors propose integrating a BDI-based agent architecture into the GAMA platform. This proposal defines a simple BDI agent architecture with many restrictions compared to other BDI implementations. The work presented in [41] proposes JaCaMo-sim, an extension of the JaCaMo platform that allows for the simulation of MAS systems using BDI agents written in Jason. Another similar work is presented in [42], where the SAVI architecture (Simulated Autonomous Vehicle Infrastructure) is proposed. This architecture integrates multi-agent systems using the BDI paradigm with a simulation platform. Similarly, [43] introduces the Basta platform (BDI-based architecture of simulated traffic agents), which is specifically designed for the simulation of Connected and Autonomous Vehicles (CAV). The platform offers useful abstractions for most CAV activities while maintaining the goal of representing complex reasoning in a comprehensible way. Finally, the work presented [44] develops a multi-agent system for a specific domain, the Disaster–Rescue domain, by using cognitive agents written in Jason with NetLogo, the well-known agent-based simulation platform. Similarly, Jack and Repast were interconnected in the work presented in [45].

As can be seen, many recent results try to apply the benefits of using BDI agents for current real environments or in more realistic and complex simulations. However, the review has also revealed the lack of frameworks that can offer the possibility of developing complex systems (simulated or not) where BDI agents can be used in conjunction with other types of agents or technologies. Such a framework should provide facilities allowing for the integration of BDI models with other techniques, such as machine learning, data fusion, IoT standards, ecosystems interoperability, human–machine interfaces, and so on, in order to take full advantage of all types of current technologies.

3. SPADE 3

SPADE 3 [46] is the latest version of the SPADE middleware (https://github.com/javipalanca/spade, accessed on 7 January 2023), which supports the development and execution of multi-agent systems. The main underlying idea of SPADE is to build multi-agent

systems around a well-established, standard communication protocol called XMPP [47]. XMPP is an open and extensible protocol for instant messaging and presence notification that is used in many human-to-human communication applications nowadays. Hence, SPADE proposes the MAS developer design the multi-agent system from the perspective of a modern, typically human, communication model. In addition, and because of how XMPP has been designed, SPADE supports a wide variety of communication (and execution) scenarios. According to the communication span, it supports from a few communicating entities connected to a single server to hundreds of entities connected to several servers across the Internet. According to privacy and security aspects, it supports from fully open to private deployments with strict security requirements. According to the system size, it supports from small, self-contained systems to big developments which may integrate humans and third-party software entities directly with the application agents.

From its early versions, SPADE has been implemented in Python, and proposes it as the main programming language to develop multi-agent systems (although it may incorporate agents implemented in other languages, such as the AgentSpeak extension described in this paper). In particular, SPADE 3 enforces an object-oriented and asynchronous programming model based on the *AsyncIO* library, with a fair trade-off between simplicity and scalability. The asynchronous model allows for more efficient use of the computational resources in applications where running entities alternate computation and communication activities, as it happens in multi-agent systems. On the other hand, SPADE proposes to design multi-agent applications by following a particular agent model, which is similar to the one present in other platforms (as JADE, for example). Under the SPADE agent model, each agent is internally structured in one or various behaviors, with each one being an independent executable task following a particular execution pattern that is appropriate to the task's characteristics. There are several available behavior types, each one enforcing a particular execution pattern: Cyclic, One-Shot, Periodic, Time-Out, and Finite State Machine.

Since communication is one of the key functions of SPADE agents, the internal architecture of the SPADE model is communication-centered. Figure 1 shows a diagram representing how agents are run inside a SPADE process. Each SPADE process may contain multiple agents, all of them internally managed by an *event loop*, which is a Python tool for scheduling and executing asynchronous functions. Each agent internally consists of a dispatcher and the set of behaviors that the agent has created. The dispatcher is in charge of keeping alive the connection between the agent and the XMPP server, as well as sending and receiving messages. The received messages are distributed to the particular behavior(s) to which they are addressed, using a series of filters or templates associated with each behavior (e.g., all messages with a particular sender and performative). Then, each message is stored in the mailbox of the designated behaviors until it is read. On the other hand, from this viewpoint, each behavior is a task that can be independently scheduled and executed by the process event loop, in a non-preemptive way.

In addition to the programming model, described above, the SPADE middleware offers a series of services to facilitate the development and execution of multi-agent applications. Briefly, the most important ones are now described. First, each agent must use the registering service to be known by the platform. In order to do so, the agent uses XMPP-compatible credentials (an agent identifier structured in the format "username@server", and a password). Second, the messaging system is responsible for relaying messages from any two agents in the platform, whether such agents are running on the same computer or in different ones, connected within a local network or through the Internet. In particular, SPADE associates a message dispatcher to each agent, which is responsible to deliver each incoming message to the particular behavior (or behaviors) which may be expecting it. Third, SPADE incorporates different security services which, in turn, are based on mechanisms provided by the XMPP protocol. For example, XMPP naturally provides end-to-end encryption and signed messages for all communications if needed, securing both the channel and the authenticity of the participant agents. Fourth, SPADE also provides the developer with the ability to incorporate new features and services, via

plugins, and also via XMPP extensions (called XMPP Extension Proposals or XEPs). Finally, the presence notification mechanism is another XMPP-based service by which each agent is provided with a presence status (including its current availability situation and any other interesting feature about its internal state), and SPADE automatically informs its fellow agents of every time this status changes. Since this service is particularly involved in the flexible architecture proposed in this paper, it will now be described in more detail.
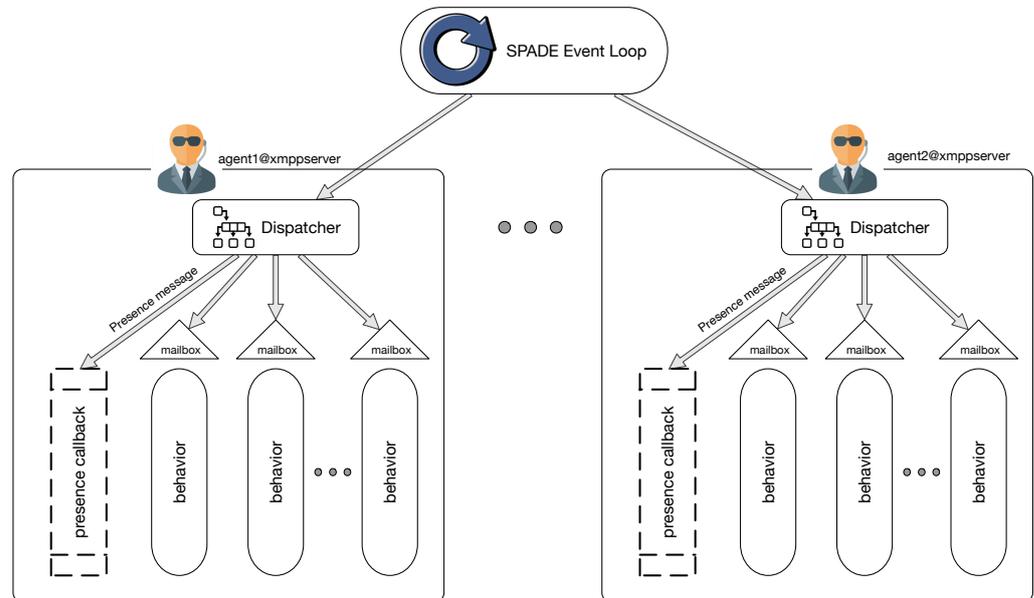

Figure 1. Internal architecture of a SPADE process.

As originally defined by XMPP, the presence notification service allows a (human) user to be aware of the availability status (available, busy, offline, etc.) of any of the user's contacts, which are other users who have previously accepted a friendship request. The presence information is not limited to communicating availability (available or unavailable). This presence information allows agents to communicate completely personalized information about the current status they wish to share. It is important to note that this information is not the agent's internal status, but the customized status information that the agent wants to share with its contacts. Thus, the presence information will be shared automatically but only with the agent's authorized contacts, the ones who belong to the list of contacts that the agent has previously approved. Adapting this idea to multi-agent systems, SPADE offers this unique feature by which agents may be notified in real-time the customized presence message of other agents in the system with whom they have previously established a "friendship" relationship. This relationship is achieved by means of a subscription mechanism, whereby an agent (*Agent A*) can request another one (*Agent B*) to subscribe to its presence notifications. If Agent B accepts the subscription, Agent A will thereafter receive presence alerts from the server whenever the presence information of Agent B changes. In addition to the pure presence information (to be online or offline), SPADE supports agents to share any specific, application-defined, information which may be useful for their contacts (such as working, going to a destination, waiting for a response, etc.). In fact, SPADE has extended this mechanism so that agents may use it to coordinate their actions by implementing sophisticated instruments, such as synchronization barriers. This way of using presence notification is an improvement in agent programming since it simplifies communication with groups of agents with whom they wish to share information (status, domain, or whatever the agent decides) in a very simple way. Thus, the agent only has to change its presence information, and then the XMPP protocol will be in charge of sending this information to all the authorized agents. On the other hand, the presence information is decided by the agent, so it does not necessarily have to be true or match

its internal state. Therefore, its semantics will depend on the application domain and the benevolence of the agent. In addition, the group of agents to which the agent shares its presence information is dynamic, since the agents belonging to an agent's contact list may be modified as subscription requests are handled.

Internally, SPADE supports presence notification by means of a special type of message, called presence notification message, and the use of callbacks. A presence notification message is a special type of message, with which a user or agent sends a packet of information containing its availability status to the server. Then, the server, which maintains the list of contacts for each agent in its persistence, redirects the presence message to all the contacts that should be notified which are actually connected at that moment, effectively minimizing the number of messages needed. A presence message consists of two fields: the availability of the user or agent (which can be "available" or "unavailable") and the status to be displayed (the *show* field), where it can specify a free text that identifies the specific availability status (busy, away, gone to lunch, etc.). It is by this second field that multi-agent applications can implement indirect communication and coordination among agents, for example.

On the other hand, SPADE proposes using callbacks to provide an efficient, reactive mechanism to receive presence notification messages, as well as the subscription and unsubscription requests to the agent's presence information. In particular, instead of delivering these messages to any agent's behaviors (and thus requiring a dedicated behavior to process them), the agent can configure a custom method, the callback, which will be automatically invoked by SPADE whenever each of these messages arrives. This way, the agent may decide in a reactive method to accept or not the subscription request (which would be the quickest response), but, in case the decision would require a more complex deliberative process, the agent could create and execute a behavior to handle it.

4. Flexible Agent Architecture

This section presents the Flexible Agent Architecture (FAA), an architecture which intends to favor the development of *flexible* multi-agent systems, allowing for the combination of all kinds of agents, from purely reactive or reflex to purely deliberative ones, and including hybrid ones. The section first introduces the definition of the architecture and then describes how it has been introduced to the SPADE 3 middleware.

4.1. Definition of the Architecture

The Flexible Agent Architecture is founded on the idea behind the behavior-based control [48], which defines a set of distributed, interacting modules, called behaviors, that collectively achieve the desired agent behavior. The idea is taken from the area of robotics, where behaviors are control modules that cluster sets of constraints in order to achieve and maintain a goal [49].

Sometimes, behavior-based architectures have been confused with reactive architectures, since historically, reactive architectures have been designed by including different reactive behaviors structured in layers, where intelligence emerges from the activation or deactivation of the outputs between the different layers. However, behavior-based architectures are conceptually more capable, since they remove some of the limitations of reactive systems. In particular, there are no restrictions on maintaining an internal representation of the environment (a traditional trait of deliberative architectures) or designing arbitrarily complex behaviors. Among other considerations, this implies that the internal state of the agent, including the world representation, may be distributed among the different behaviors. Thus, compared to reactive architectures, here behaviors distribute not only inputs and outputs, but also the internal state and the decision-making process of the agent, and some of these behaviors may be deliberative, maintaining a state and a world representation. All this greatly increases the flexibility of this type of architecture.

Taking this idea into account, the FAA proposes a behavior-based architecture in which each agent in the multi-agent system is internally defined as a set of behaviors, with

each behavior being either procedural or logic-based (as for example, following the BDI model), and all of them sharing a common internal state and a world representation. On the one hand, this proposal favors the combination of multiple types of reasoning processes in the same agent, including, but not limited to, reflex, reactive, repetitive, and logic-based processes. Additionally, on the other hand, it overcomes some significant difficulties of pure logic-based agents, such as expressing some sorts of algorithms (e.g., neural networks or genetic algorithms) or including reflex answers apart from its reasoning cycle. As a result, the multi-agent system may include agents of multiple types, from reactive (even reflex) agents to pure deliberative agents, and any sort of hybrid ones. As the name of the proposal implies, it has to be underlined the flexibility provided if compared with traditional hybrid-agent architectures, either horizontal, like TouringMachines [50], or vertical, like InterRap [51]. The proposal based on behaviors is less rigid than traditional hybrid ones since behaviors do not have the strict structure that is typical of such architectures.

Some existing agent platforms, such as SPADE (or JADE), have been designed to be behavior-based, in the sense that agents are developed by implementing one or several behaviors, each of them following some pre-defined execution pattern (one-shot, cyclic, etc.) So, the implementation of the FAA in SPADE has mainly involved including BDI behaviors to agents, as described below.

4.2. Implementation in SPADE 3

As expressed above, the Flexible Agent Architecture offers a versatile framework that facilitates the development of multi-agent systems where each agent may require reactive or deliberative capabilities or both. A version of this architecture has been implemented as an extension to the SPADE 3 middleware (https://github.com/javipalanca/spade_bdi, accessed on 7 January 2023), and it is shown in Figure 2.
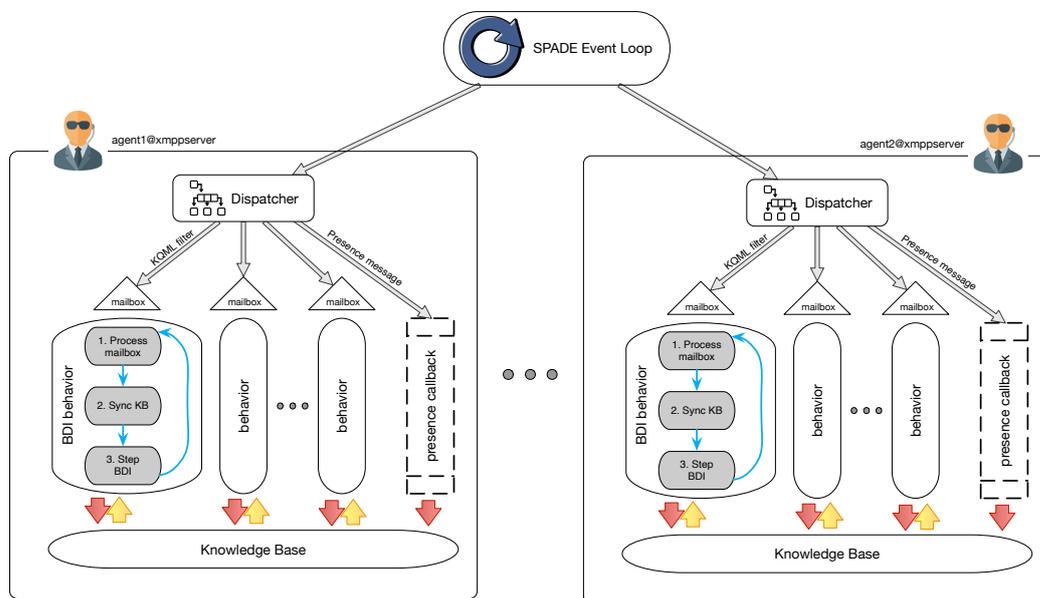

Figure 2. Flexible Agent Architecture in SPADE 3.

This extension incorporates a new type of behavior based on the BDI paradigm, called *BDIBehavior*, which can be adopted by any agent with deliberative requirements. The new BDI behavior type is automatically managed by SPADE, so the developer does not have to intervene to program, or keep track of, its BDI life cycle. Any SPADE agent which wants to use the BDI behavior has to create it first by providing the name of a file written in AgentSpeak format which contains the behavior's code. When the agent starts the behavior, it initiates a cyclic execution pattern in which SPADE runs its *BDI cycle* step by step, as is further described below. This cyclic execution is carried out indefinitely, and concurrently with any other behaviors that the agent may create. Although the behavior is automatically

executed by SPADE, the agent still has some control over its execution, by pausing or resuming it at any time.

Internally, a BDI behavior includes a modified version of an AgentSpeak interpreter that has been integrated into SPADE 3. This modified interpreter includes some features that increase its capabilities, such as the ability of AgentSpeak actions to directly call SPADE 3 services and/or methods implemented in Python. This extends the set of actions that may be developed with AgentSpeak to any custom (or library) method which can be implemented in Python and executed in SPADE. As a result, the developer is enabled to implement each action in ASL or directly in Python (and, in this latter case, to have a vast set of library methods), depending on which option better suits the needs of the application.

Another important feature of the BDI behavior is the set of communication skills that have been incorporated into the interpreter, in order to facilitate its integration and information exchange with other behaviors and agents. There are two different communication methods, depending on whether the interaction of the BDI behavior is internal to the agent (that is, to other behaviors in the same agent) or to other agents in the system. In the first case, the communication is carried out by providing a common knowledge base in the agent that all the behaviors share. By means of this knowledge base, the concept of Belief in the BDI behavior has been extended outside the AgentSpeak interpreter, allowing any agent's behaviors to access, insert, and erase beliefs. Thus, the set of beliefs is global to the agent, and the modification of beliefs (by any behavior) produces the corresponding events in the interpreter, ensuring a consistent response in the ASL language. On the other hand, the communication of the BDI behavior with other agents (and their behaviors, BDI or not) is performed by directly using the SPADE 3 communication facilities. Such facilities have been incorporated into the interpreter, enabling transparent communication between the BDI behaviors of any two SPADE agents by using the natural ASL syntax. This communication has been carried out using KQML messages, as this communication language is the source of AgentSpeak communication [52]. KQML implements the key concept of performative, which allows the sender to influence the receiver in different ways according to the *speech act theory* [53].

Communication between the reactive behaviors and the BDI behavior occurs through the Knowledge Base (KB), which is a common space that all behaviors can access (the BDI behavior natively and the rest of the behaviors by means of methods that query or modify the KB). Thus, access to the agent's environment from the BDI behavior is provided both by the messages received and by the KB, which can be modified by any behavior that receives a stimulus both from the outside and from its internal processes.

In addition, the presence notification mechanism has also been considered in the implementation of the FAA in SPADE. In essence, the implementation provides every type of behavior with the ability to access the presence information about the agent's contacts. In particular, a reactive behavior will use the traditional way of registering a callback method in order to react to presence updates of the agent's contacts (previously described in Section 3). On the other hand, a BDI behavior will be provided with the belief that represents the availability (presence) status of each of the agent's contacts. This belief will be automatically updated by SPADE whenever the presence status of the contact changes. This is represented in Figure 2 as a red arrow from the presence callback to the knowledge base, meaning that after processing any callback, the presence information will also be injected into the knowledge base. As a result, these presence information beliefs can be naturally integrated into any BDI plan. For example, a particular plan could be triggered when one (or several) of the agent's contacts become 'available', and not before.

Considering the agent's own presence status, it can be changed whenever necessary from either a reactive behavior or a BDI behavior, so that the agent's contacts will be notified. From a reactive behavior, this is performed by calling the `set_presence` method, while from BDI, the status can be changed by executing the internal action `.set_presence`. Finally, the presence subscription mechanism by which an agent can accept (or not) a "friendship" request from another agent has not been integrated into BDI behaviors. Since

this interaction between agents is a kind of request-response performative, the handling of this interaction by a reactive behavior is considered a more straightforward solution.

In conclusion, the implementation of this flexible architecture in SPADE 3 allows for consistent integration of both BDI and non-BDI behaviors in any SPADE agent, unlike the proposals found in the literature that have been commented upon in Section 2. The integration is twofold: on the one hand, non-BDI behaviors (implemented in Python) can access the BDI knowledge base by reading and modifying beliefs and objectives; in the latter case, the modifications produce the same effect as if they were performed inside the BDI reasoning process. On the other hand, BDI behaviors (expressed in AgentSpeak language) may call actions implemented in Python, which enable them to access the rest of the behaviors of the agent, as well as the agent's context outside BDI, that is, the agent's global information which is not represented as beliefs or objectives.

5. Case Study

This section presents an example illustrating the use of the proposed Flexible Agent Architecture in a scenario where applying deliberative and non-deliberative tasks presents an advantage. In particular, the section introduces a multi-agent system in the context of a factory where different robot arms collaborate to move items throughout the manufacturing process. The application has been designed as a simplified object management problem combining a BDI behavior in charge of the reasoning process of each robotic arm, and other behaviors in charge of the vision and movement of the arm. In addition, the coordination among the robotic arms is managed by the BDI behavior using the presence mechanism offered by SPADE. Moreover, the environment has been created by means of an external 3D simulator.

The multi-agent system of the example has been designed as a client-server application, where robot arms are modeled as agents, and their coordination is obtained by means of the Presence Notification feature. Each one of these agents is internally designed as a set of collaborating behaviors and then implemented in the SPADE platform enhanced with FAA, as described earlier in the paper. In particular, each agent's reasoning process is implemented as a BDI behavior (following the AgentSpeak specification). Other tasks of the agents, including interfacing with the robot arm and the perception of the environment, have been implemented (in Python) through additional behaviors and modeled as actions inside the BDI behavior.

The following subsections separately describe the case study details, the simulated environment, and the implementation of the multi-agent system in SPADE. Then, a final subsection presents some discussion about the development process and the resulting system.

5.1. Description of the Case Study

The case study presented here is a simplified version of a factory operated by robots, in which a producer robot places pieces of different types (or shapes) on a shared workspace. Then some consumer robots fetch these pieces and classify them according to their types.

The environment (see Figure 3) consists of three rectangular tables placed in an L-shaped layout (labeled A, B, and C) and three robotic arms (*arm_1*, *arm_2*, and *arm_3*) that can pick up objects by using a suction pump. Each of the three arms is attached to a table and it can only reach its own table and the adjacent one. The *arm_2* that is attached to the middle table (B) is the one providing the pieces (that is, placing them on the middle table). The *arm_1* and *arm_3* are the ones in charge of fetching the pieces and dropping them in the proper place according to their shapes. These three arms simulate a small production line where they can collaborate with a human. Each robot integrates a camera located in the end effector, which is used as an input device to perceive the environment and helps the robot detect objects within its range of vision. The images captured by the camera are used for locating the objects and detecting their shapes. For the sake of simplicity, the system for this case study is focused on detecting basic geometric shapes. However, the versatility of the proposed architecture would easily permit the integration of deep

learning models for recognizing more complex objects, as well as introducing new objects in real-time, producing a more agile and versatile production chain.
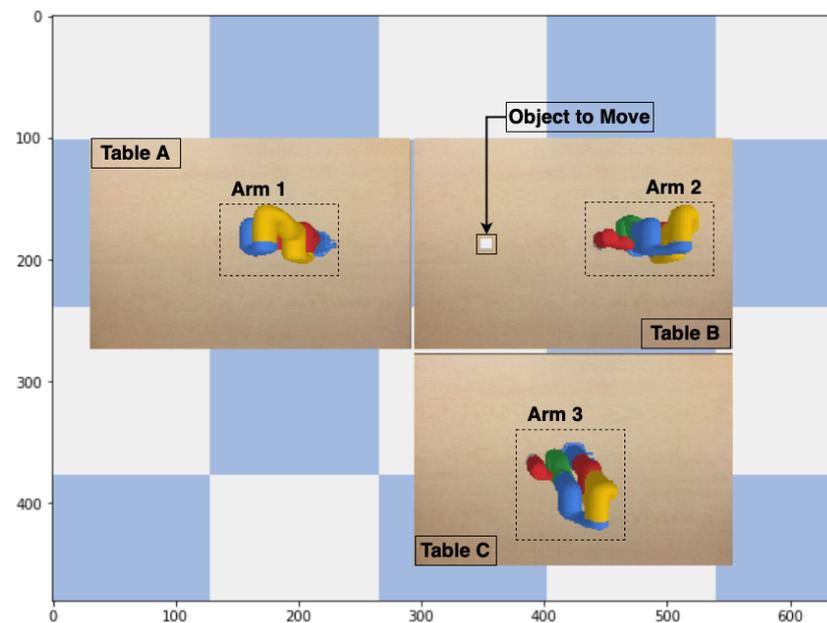

Figure 3. Environment setup for the case study.

The environment was built using a 3D simulation tool called PyBullet (https://pybullet.org/wordpress/, accessed on 7 January 2023). In the simulation environment, the three robotic arms to be operated by the multi-agent application are real, commercial robots called Panda arms (https://www.pomorobotics.com/robots/frankpanda/, accessed on 7 January 2023). These Panda arms feature 7 degrees of freedom, which will require the application agents to perform complex actions in order to control them, ensuring a realistic interaction with the environment. In addition, the environment includes cameras as perception (input) devices for the application agents. There are three cameras, each focused on one of the tables and providing a top (zenithal) view of this table. By analyzing the images taken by a camera, an application agent will be able to locate the object placed on its corresponding table, and then move the arm to that location to pick that object up. So, the application agents will interact with the environment by sending commands to the PyBullet server, such as *"take an image from the zenithal camera"*, *"move the arm to a coordinate"*, *"use the vertical suction pump to pick up an object"*, etc.

Figure 4 presents a diagram of the multi-agent system architecture designed for the case study, depicting the application agents and the PyBullet server. As can be seen in the figure, a decentralized application is proposed, where there are three SPADE agents, each one controlling one robotic arm, and each one executed in a separate process (and potentially from a different computer), while a separate (fourth) computer runs the PyBullet server. The separation between the physical simulation and the agents enables the application to interact with either the simulated or a real environment without changing the code executed by the agents. The three agents can coordinate with each other (in order to achieve the goal of providing pieces and picking up and dropping them in their place) by means of the Presence Notification and the inter-agent communication service of SPADE (which internally uses the XMPP protocol). Communicating with the PyBullet server is made directly by calling the corresponding API functions.
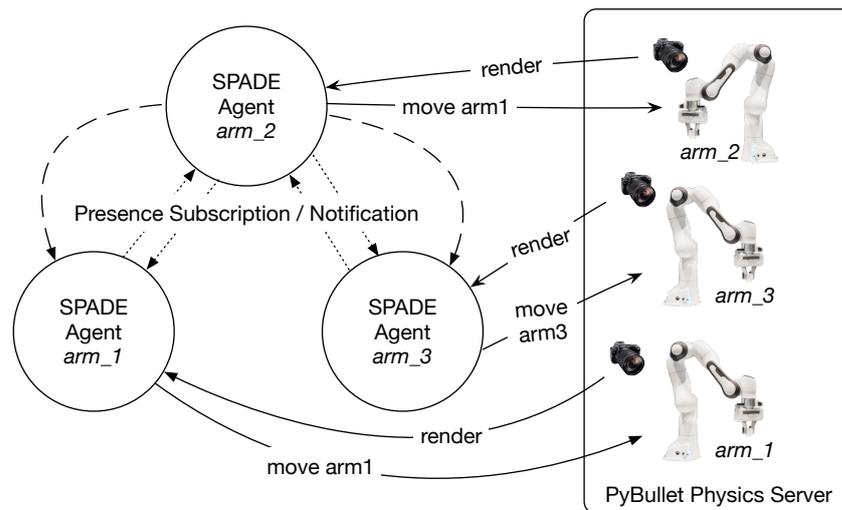
Figure 4. Architecture of the case study. Dashed lines denote message exchange using the SPADE communication services while solid lines denote API function calls to the PyBullet server. Dotted lines denote Presence Notification messages.

Thus, the multi-agent system implemented in SPADE for the case study consists of three agents, each one controlling a robotic arm, which communicate with each other in order to achieve the common goal of solving the producer/consumer scenario. Agents are named after the arm each one operates, i.e., *arm1*, *arm2*, and *arm3*. Internally, each agent has been designed with two main behaviors: a BDI behavior in charge of making the agent's high-level decisions, and an FSM (Finite State Machine) behavior which is the one interacting with the (simulated) environment. These two behaviors cooperate with each other by sharing beliefs in the agent's knowledge base. Figure 5 presents a schematic view of both behaviors, which are now described in the following subsections.
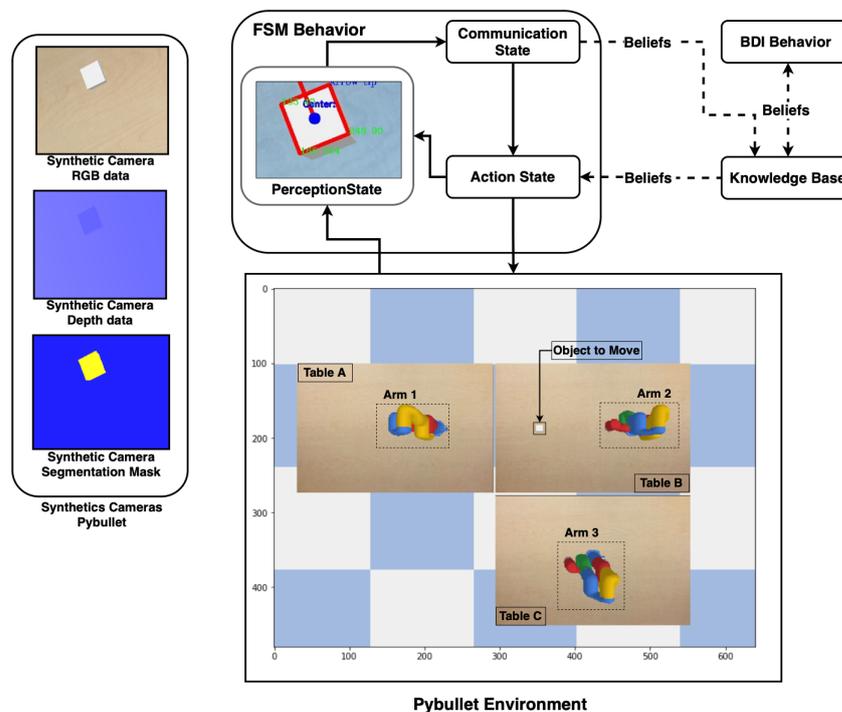

Figure 5. Diagram of the two behaviors inside each of the SPADE agents, including their interactions with each other and the environment: solid lines represent the transitions between states in the FSM behavior, while dashed lines represent information exchange among the agent's behaviors, through the agent's knowledge base.

5.2. Interaction (FSM) Behavior

The *Interaction* behavior is an FSM behavior composed of three states, called Perception, Communication, and Action. The Perception state interacts with the PyBullet API in order to retrieve images from the overhead camera focused on the table/arm controlled by the agent. Specifically, the PyBullet server returns three types of images from the camera (a segmented image, a raw image, and a depth image), from which the agent can extract relevant information from the environment, such as contour detection, shapes, segmentation, etc. An example of this segmentation process carried out by the Perception state can be seen in the left part of Figure 5. This figure also shows how the agent uses the images to determine the position of the object's centroid and its outline, which allows it to delimit the location of the piece. It then transforms all these data into the precise location of the piece within the table under the camera.

In particular, in order to detect objects through the camera, the agents use the Open CV2 library. OpenCV is an open-source library mainly used to process images and videos in order to identify shapes, objects, text, etc. It is mainly used with Python. For the robots to be able to detect shapes in the captured image, a transformation of the image colors to greyscale is necessary. This new grayscale image is then transformed into a binary image (black and white), which makes the figure a soft color against a black background. Then, the next step is the search for the object's contour, which is the boundary of the shape to be detected within the image. Once the outline of the object is delimited, the next step is the extraction of the object's centroid, which determines the coordinates where the object is located. Then, from the coordinates of both the object and the arm's effector, it is possible to calculate the movement of the effector in order to capture the object.

So, the Perception process interacts with the agent's environment by using the artificial vision process described above. This process, due to its complexity, is implemented in a reactive behavior. Then, it modifies the agent's knowledge base with the information captured from the environment, so that the deliberative (BDI) behavior can reason about this information and make decisions about the next actions to be performed.

Once the Perception is complete, the Communication state then transforms the significant information about the environment (basically, the object's location) to a format that is appropriated to be processed by the BDI behavior, and writes that information as *beliefs* in the agent's knowledge base. The BDI behavior, as described below, reasons about it and decides which is the next action to be taken by the agent controlling the arm. That decision, also written as beliefs in the common knowledge base, is read by the Action state, which then interacts with the PyBullet server in order to send the next command (move the arm, pick the object up, take it to a particular position, etc.), exactly as if it was a real Panda arm.

5.3. Deliberation (BDI) Behavior

The *Deliberation* behavior is a BDI behavior including a set of plans which, according to the perception of the FSM behavior, its internal state, and its current objectives, decides the next action to be carried out by its respective robotic arm.

As commented above, *AgentSpeak* is the logic programming language provided by SPADE in order to implement BDI behaviors. As a result, the beliefs of an agent that features a BDI behavior, including its internal state and its environment (or world) representation, are represented as predicates in *AgentSpeak*'s predicate logic. For example, in the case study, the belief `object(type).` represents (by means of a predicate) that, according to the agent's information, the object with the shape `type` is currently caught by the robot's pump.

In the same way, the *desires* of agents are represented as goals, that is, as predicates that the agents want to fulfill (in other words, to be part of the beliefs). Syntactically, *AgentSpeak* expresses desires as predicates preceded by the `!` symbol. For example, in the case study, the desire `!move_object(tableA, tableB)` in one of the agents would mean that one of the agent's goals is to move an object from the table named `tableA` to table named `tableB`.

Finally, *AgentSpeak* allows for the definition of plans to answer events happening to the agent, for example, events related to achieving new goals. Thus, the *intentions* of agents

are defined by the plans which are instanced to react to those events. For example, the following code shows a subset of the implemented plans for the SPADE agents to define the decision-making of the agents. On the one hand, this code shows how the agents coordinate with each other by using the presence mechanism. On the other hand, the code also shows the plans that allow an agent to deposit objects with different shapes while, at the same time, other robotic arms are in charge of picking them up and dropping them in the correct positions in their respective tables.

```
1   ...
2   // **************************************************
3   // Robot in charge of putting the objects
4   // **************************************************
5   // Init
6   +start
7   <-
8       .set_presence("IDLE");
9
10  // Tries to get an object from table T
11  +!move_object(T, T1) :  .my_name(N) & status(N, "IDLE")
12  <-
13      .pick_up_object(T);
14      .set_presence("BUSY");
15      -+destination(T1).
16
17  // object picked and move to table T1
18  +object : not status( _ , "TAKING") & status(X, "IDLE")  & destination(T1)
19          <-
20      .drop_object(T1);
21      .set_presence("IDLE");
22      .send(X, achieve, pick_up(T1)).
23
```

These first three plans represent the decision-making knowledge of the robotic arm in charge of picking up objects of any shape and dropping them on an intermediate table (*arm2* in this particular example). Initially, it sets its presence state to "IDLE". Subsequently, when there is an object to pick up, the `move_object` objective is triggered. Then, if the agent is not busy, it will pick up the object (through the action `.pick_up_object(T)`) and change its presence state. The last plan reflects the implicit coordination involved between the robotic arms since the agent will only drop the object on the exchange table when the rest of the robots are not picking an object and there is at least one free arm. In that case, the agent will try to drop the object on the corresponding table (`.drop_object(T1)`) and will notify the robot arm that was waiting to pick up the object.

The next plans represent the decision-making knowledge of the robotic arms in charge of sorting objects into different tables according to their shapes (*arm1* and *arm3* in the example).

```
1   // **************************************************
2   //  Robot in charge of picking up the objects
3   // **************************************************
4   // Init
5   +start
6   <-
7       .set_presence("IDLE");
8
9   // Tries to get an object from table T
10  +!pick_up(T) :  .my_name(N) & status(N, "IDLE")
11      <-
12      .set_presence("TAKING");
13      .pick_up_object(T).
14
15  // object picked and move to the correct position in its table
16  +object(Type) : destination(Type, Pos)
17  <-
18  .drop_object(Pos);
19
20   // changes state when the arm no longer has the object.
21  -object(_) :
22      <-
23      .set_presence("IDLE");
24
25  ...
```

With this code, like the producer agent above, the agent initializes its presence state to "IDLE". Then, when it receives a request to pick up an object, it checks that its state is "IDLE" and picks up the object with the action `pick_up_object(T)`. When it picks up the object, a belief of type `object(Type)` is activated where `Type` indicates the object's shape. This belief, in turn, triggers a plan which, according to the object's shape, moves the object to the corresponding position on its table with the action `.drop_object(Pos)`. At the end of the process, it updates its presence state again.

5.4. Discussion

The case study presented in this section aims to illustrate the flexibility of the proposed architecture and its implementation in SPADE with a simple but well-known example. By using the different levels of abstraction presented in this architecture, it is possible to solve different aspects, such as the combination of BDI and non-BDI behaviors for the decision-making in the same agent, as well as the exchange of information by means of messages or presence notifications between agents, and a simple integration with a close-to-real simulated environment which is easily scalable. In this sense, the proposed FAA has made it easy the creation of a robust and dynamic application. As it has been presented, it allows for fast and straightforward integration and interaction between the BDI cycle and other behaviors inside each agent, helping the developer to integrate sensors, actuators, and other reasoning engines.

It should be noted that the case study has been specifically designed to illustrate the use of the FAA and, in this sense, some aspects of a real system have been simplified. For example, although the example only makes use of three agents, it can be easily adapted to environments that require more robotic arms, a different disposition of the tables and robots, robotic arms of different types, etc.

6. Conclusions

The rising interest in some application domains, such as cyber–physical systems, the Internet of Things, or self-driving vehicles, among others, has renewed the attention to multi-agent systems (MAS) as a solid, useful technology capable of dealing with complex and dynamic environments. In particular, MAS following behavior-based architectures are well fitted for providing fast, reliable responses while also maintaining an internal world representation by which agents may learn from experience, in order to adapt their outcomes to the dynamics of the environment. In this context, this paper has introduced the Flexible Agent Architecture (FAA), with the goal of integrating BDI reasoning capabilities as a full-fledged behavior in an agent. By doing so, agents can combine all types of cognitive and reactive reasoning processes, effectively enhancing their application range.

The Flexible Agent Architecture has been applied to the SPADE platform. SPADE was originally designed as a behavior-based platform, with a variety of behaviors available to each agent in order to define its decision-making process. By following this new FAA, agents can also use a BDI-based behavior founded on the *AgentSpeak* BDI model in order to improve their deliberative reasoning processes. The design of this *BDI behavior* has considered both the programming model and also the services available in SPADE, in order to achieve seamless integration. This includes three main features. First, each agent is provided with a common knowledge base for all its behaviors (BDI or not), in order to allow for a natural communication mechanism among such behaviors. Second, BDI behaviors are provided with a communication mechanism in terms of the AgentSpeak language, which uses the SPADE underlying messaging facilities. In addition, KQML messages are specifically supported in order to facilitate the direct communication between BDI behaviors of different agents. Third, BDI behaviors are also provided with presence notification capabilities as any other SPADE behavior. This includes not only the possibility of changing the agent's own presence status from BDI but also integrating the presence information of the agent's contacts directly into the BDI reasoning cycle. As a result of all this, SPADE can implement purely reactive agents (implemented with fast reactive

behaviors), deliberative agents (implemented with a BDI behavior), and hybrid agents (implemented with both reactive and BDI behaviors). This obviously upgrades the ability of SPADE to develop MAS which can be used in several domains.

The paper also includes a case of study where this new flexible architecture has been applied to develop an actual multi-agent system in SPADE. The system controls a simplified factory scenario in which three robotic arms need to coordinate with each other in order to detect and move pieces of different shapes. The environment has been simulated by means of a realistic 3D simulator which allows for a quick transfer from the simulation to reality. The presented multi-agent system is intentionally simple in order to illustrate the facility with which a BDI reasoning cycle can be integrated with other agent processes in a completely parallel manner; however, it could easily be extended to solve a much more challenging problem in the same scenario, thanks to the capabilities of the BDI reasoning model.

Lastly, the authors would like to underline that the work presented here is freely available to download and be used in PyPI (https://pypi.org/project/spade-bdi/, accessed on 7 January 2023), and its sources available in GitHub (https://github.com/javipalanca/spade_bdi, accessed on 7 January 2023).

Author Contributions: Conceptualization, J.P. and A.T.; methodology, C.C.; software, J.P. and J.A.R.; validation, V.J.J.; formal analysis, V.J.J.; investigation, J.P. and A.T.; resources, J.A.R.; writing—original draft preparation, J.P., J.A.R., C.C. and V.J.J.; writing—review and editing, A.T.; supervision, V.J.J. All authors have read and agreed to the published version of the manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Suganuma, T.; Oide, T.; Kitagami, S.; Sugawara, K.; Shiratori, N. Multiagent-based flexible edge computing architecture for IoT. *IEEE Netw.* **2018**, *32*, 16–23. [CrossRef]
2. Belkhala, S.; Benhadou, S.; Boukhdir, K.; Medromi, H. Smart parking architecture based on multi agent system. *Int. J. Adv. Comput. Sci. Appl.* **2019**, *10*, 378–382. [CrossRef]
3. Howell, S.; Rezgui, Y.; Hippolyte, J.L.; Jayan, B.; Li, H. Towards the next generation of smart grids: Semantic and holonic multi-agent management of distributed energy resources. *Renew. Sustain. Energy Rev.* **2017**, *77*, 193–214. [CrossRef]
4. Cao, J.; Bu, Z.; Wang, Y.; Yang, H.; Jiang, J.; Li, H.J. Detecting prosumer-community groups in smart grids from the multiagent perspective. *IEEE Trans. Syst. Man Cybern. Syst.* **2019**, *49*, 1652–1664. [CrossRef]
5. Calvaresi, D.; Marinoni, M.; Sturm, A.; Schumacher, M.; Buttazzo, G. The challenge of real-time multi-agent systems for enabling IoT and CPS. In Proceedings of the International Conference on Web Intelligence, Leipzig, Germany, 23–26 August 2017; pp. 356–364.
6. Chang, K.C.; Chu, K.C.; Wang, H.C.; Lin, Y.C.; Pan, J.S. Agent-based middleware framework using distributed CPS for improving resource utilization in smart city. *Future Gener. Comput. Syst.* **2020**, *108*, 445–453. [CrossRef]
7. Yasin, J.N.; Mohamed, S.A.; Haghbayan, M.H.; Heikkonen, J.; Tenhunen, H.; Plosila, J. Navigation of autonomous swarm of drones using translational coordinates. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Trustworthiness, Proceedings of the 18th International Conference on Practical Applications of Agents and Multi-Agent Systems, L'Aquila, Italy, 7–9 October 2020*; Springer: Cham, Switzerland, 2020; pp. 353–362.,
8. Schaefer, M.; Vokřínek, J.; Pinotti, D.; Tango, F. Multi-agent traffic simulation for development and validation of autonomic car-to-car systems. In *Autonomic Road Transport Support Systems*; Springer: Cham, Switzerland, 2016; pp. 165–180.
9. Guastella, D.A.; Camps, V.; Gleizes, M.P. Multi-agent Systems for Estimating Missing Information in Smart Cities. In Proceedings of the 11th International Conference on Agents and Artificial Intelligence (ICAART 2019), Prague, Czech Republic, 19–21 February 2019; pp. 214–223.
10. Fortino, G.; Fotia, L.; Messina, F.; Rosaci, D.; Sarné, G.M. A meritocratic trust-based group formation in an IoT environment for smart cities. *Future Gener. Comput. Syst.* **2020**, *108*, 34–45. [CrossRef]
11. Pal, C.V.; Leon, F.; Paprzycki, M.; Ganzha, M. A Review of Platforms for the Development of Agent Systems. *arXiv* **2020**, arXiv:2007.08961.
12. Rao, A.S. AgentSpeak (L): BDI agents speak out in a logical computable language. In *Agents Breaking Away, Proceedings of the European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, 22–25 January 1996*; Springer: Berlin/Heidelberg, Germany, 1996; pp. 42–55.

13. Rao, A.S.; Georgeff, M.P. BDI agents: From theory to practice. In Proceedings of the First International Conference on Multiagent Systems, San Francisco, CA, USA, 12–14 June 1995; Volume 95, pp. 312–319.

14. Bratman, M. *Intention, Plans, and Practical Reason*; Harvard University Press: Cambridge, MA, USA, 1987; Volume 10.

15. Miled, A.B.; Dhaouadi, R.; Mansour, R.F. Knowledge Deduction and Reuse Application to the Products' Design Process. *Int. J. Softw. Eng. Knowl. Eng.* **2020**, *30*, 217–237. [CrossRef]

16. Shoham, Y. AGENT0: A Simple Agent Language and Its Interpreter. In Proceedings of the AAAI-91, Anaheim, CA, USA, 14–19 July 1991; pp. 704–709.

17. Braubach, L.; Lamersdorf, W.; Pokahr, A. JADEX: Implementing a BDI-Infrastructure for JADE Agents. *EXP Search Innov. (Spec. Issue on JADE)* **2003**, *3*, 77–85.

18. Nunes, I.; Lucena, C.; Luck, M. BDI4JADE: A BDI layer on top of JADE. In Proceedings of the Workshop on Programming Multiagent Systems, Taipei, Taiwan, 3 May 2011; pp. 88–103.

19. Bellifemine, F.; Poggi, A.; Rimassa, G. JADE—A FIPA-compliant agent framework. In Proceedings of the PAAM-99, London, UK, 19–21 April 1999; Volume 99, p. 33.

20. Albayrak, S.; Wieczorek, D. JIAC–An Open and Scalable Agent Architecture. *Intell. Agents Telecommun. Appl. Basics Tools Lang. Appl.* **1998**, *36*, 189.

21. Busetta, P.; Rönnquist, R.; Hodgson, A.; Lucas, A. Jack intelligent agents-components for intelligent agents in java. *AgentLink News Lett.* **1999**, *2*, 2–5.

22. Dastani, M. 2APL: A practical agent programming language. *Auton. Agents Multi-Agent Syst.* **2008**, *16*, 214–248. [CrossRef]

23. Hindriks, K.V.; De Boer, F.S.; Van der Hoek, W.; Meyer, J.J.C. Agent programming in 3APL. *Auton. Agents Multi-Agent Syst.* **1999**, *2*, 357–401. [CrossRef]

24. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason*; John Wiley & Sons: Hoboken, NJ, USA, 2007; Volume 8.

25. Aschermann, M.; Dennisen, S.; Kraus, P.; Müller, J.P. LightJason, a Highly Scalable and Concurrent Agent Framework: Overview and Application (Demonstration). In Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), Stockholm, Sweden, 10–15 July 2018; Dastani, M., Sukthankar, G., Andre, E., Koenig, S., Eds.; International Foundation for Autonomous Agents and Multiagent Systems: Richland, SC, USA, 2018; pp. 1794–1796.

26. Fichera, L.; Messina, F.; Pappalardo, G.; Santoro, C. A Python framework for programming autonomous robots using a declarative approach. *Sci. Comput. Program.* **2017**, *139*, 36–55. [CrossRef]

27. Dal Moro, D.; Robol, M.; Roveri, M.; Giorgini, P. A Demonstration of BDI-Based Robotic Systems with ROS2. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complex Systems Simulation. The PAAMS Collection, Proceedings of the 20th International Conference on Practical Applications of Agents and Multi-Agent Systems, L'Aquila, Italy, 13–15 July 2022*; Springe: Cham, Switzerland, 2022; pp. 473–479.

28. Traldi, A.; Bruschetti, F.; Robol, M.; Roveri, M.; Giorgini, P. Real-Time BDI Agents: A model and its implementation. *arXiv* **2022**, arXiv:2205.00979.

29. Rafalimanana, H.F.; Razafindramintsa, J.L.; Cherrier, S.; Mahatody, T.; George, L.; Manantsoa, V. Jason-RS, A Collaboration Between Agents and an IoT Platform. In *Machine Learning for Networking, Proceedings of the International Conference on Machine Learning for Networking, Paris, France, 3–5 December 2019*; Springer: Cham, Switzerland, 2019; pp. 403–413.

30. Rafalimanana, H.F.; Razafindramintsa, J.L.; Ratovondrahona, A.J.; Mahatody, T.; Manantsoa, V. Publish a Jason agent BDI capacity as web service REST and SOAP. In Proceedings of the International Conference on the Sciences of Electronics, Technologies of Information and Telecommunications, Maghreb, Tunisia, 18–20 December 2018; Springer: Cham, Switzerland, 2018; pp. 163–171.

31. Jarvis, D.; Jarvis, J.; Yang, C.W.; Sinha, R.; Vyatkin, V. Janus: A Systems Engineering Approach to the Design of Industrial Cyber-Physical Systems. In Proceedings of the 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), Helsinki, Finland, 22–25 July 2019; Volume 1, pp. 87–92.

32. Alzetta, F.; Giorgini, P.; Marinoni, M.; Calvaresi, D. RT-BDI: A Real-Time BDI Model. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Trustworthiness. The PAAMS Collection, Proceedings of the International Conference on Practical Applications of Agents and Multi-Agent System, L'Aquila, Italy, 7–9 October 2020*; Springer: Cham, Switzerland, 2020; pp. 16–29.

33. Jiang, H.; Vidal, J.M.; Huhns, M.N. EBDI: An architecture for emotional agents. In Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems, Honolulu, HI, USA, 14–18 May 2007; pp. 1–3.

34. Sánchez, Y.; Coma, T.; Aguelo, A.; Cerezo, E. ABC-EBDI: An affective framework for BDI agents. *Cogn. Syst. Res.* **2019**, *58*, 195–216. [CrossRef]

35. Ortiz-Hernández, G.; Guerra-Hernández, A.; Hübner, J.F.; Luna-Ramírez, W.A. Modularization in Belief-Desire-Intention agent programming and artifact-based environments. *PeerJ Comput. Sci.* **2022**, *8*, e1162. [CrossRef]

36. Larsen, J.B. Going beyond BDI for agent-based simulation. *J. Inf. Telecommun.* **2019**, *3*, 446–464. [CrossRef]

37. Jensen, A.S.; Dignum, V.; Villadsen, J. The AORTA architecture: Integrating organizational reasoning in Jason. In *Engineering Multi-Agent Systems, Proceedings of the International Workshop on Engineering Multi-Agent Systems, Paris, France, 5–6 May 2014*; Springer: Cham, Switzerland, 2014; pp. 127–145.

38. Jensen, A.S.; Dignum, V.; Villadsen, J. A framework for organization-aware agents. *Auton. Agents Multi-Agent Syst.* **2017**, *31*, 387–422. [CrossRef]

39. Singh, D.; Padgham, L.; Logan, B. Integrating BDI agents with agent-based simulation platforms. *Auton. Agents Multi-Agent Syst.* **2016**, *30*, 1050–1071. [CrossRef]

40. Taillandier, P.; Bourgais, M.; Caillou, P.; Adam, C.; Gaudou, B. A BDI agent architecture for the GAMA modeling and simulation platform. In *Multi-Agent Based Simulation XVII, Proceedings of the International Workshop on Multi-Agent Systems and Agent-Based Simulation, Singapore, 10 May 2016*; Springer: Cham, Switzerland, 2016; pp. 3–23.

41. Ricci, A.; Croatti, A.; Bordini, R.; Hübner, J.; Boissier, O. Exploiting Simulation for MAS Programming and Engineering—The JaCaMo-sim Platform. In Proceedings of the 8th International Workshop on Engineering Multi-Agent Systems (EMAS 2020), Auckland, New Zealand, 8–9 May 2020.

42. Davoust, A.; Gavigan, P.; Ruiz-Martin, C.; Trabes, G.; Esfandiari, B.; Wainer, G.; James, J. An Architecture for Integrating BDI Agents with a Simulation Environment. In Proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS 2019), Montreal, QC, Canada, 13–14 May 2019; pp. 1–16.

43. Rüb, I.; Dunin-Keplicz, B. BASTA: BDI-based architecture of simulated traffic agents. *J. Inf. Telecommun.* **2020**, *4*, 440–460. [CrossRef]

44. Ramirez, W.A.L.; Fasli, M. Integrating NetLogo and Jason: A disaster-rescue simulation. In Proceedings of the 2017 9th Computer Science and Electronic Engineering (CEEC), Colchester, UK, 27–29 September 2017; pp. 213–218.

45. Padgham, L.; Scerri, D.; Jayatilleke, G.; Hickmott, S. Integrating BDI reasoning into agent based modeling and simulation. In Proceedings of the 2011 Winter Simulation Conference (WSC), Phoenix, AZ, USA, 11–14 December 2011; pp. 345–356.

46. Palanca, J.; Terrasa, A.; Julian, V.; Carrascosa, C. SPADE 3: Supporting the New Generation of Multi-Agent Systems. *IEEE Access* **2020**, *8*, 182537–182549. [CrossRef]

47. Saint-Andre, P. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, RFC Editor. 2011. Available online: https://www.rfc-editor.org/rfc/rfc6120 (accessed on 7 January 2023).

48. Michaud, F.; Nicolescu, M. Behavior-based systems. In *Springer Handbook of Robotics*; Springer: Cham, Switzerland, 2016; pp. 307–328.

49. Arkin, R.C. *Behavior-Based Robotics*; MIT Press: Cambridge, MA, USA, 1998.

50. Ferguson, I.A. Touring machines: Autonomous agents with attitudes. *Computer* **1992**, *25*, 51–55. [CrossRef]

51. Fischer, K.; Müller, J.P.; Pischel, M. A pragmatic BDI architecture. In *Intelligent Agents II Agent Theories, Architectures, and Languages, Proceedings of the International Workshop on Agent Theories, Architectures, and Languages, Montreal, Canada, 19–20 August 1995*; Springer: Berlin/Heidelberg, Germany, 1996; pp. 203–218.

52. Vieira, R.; Moreira, Á.F.; Wooldridge, M.; Bordini, R.H. On the formal semantics of speech-act based communication in an agent-oriented programming language. *J. Artif. Intell. Res.* **2007**, *29*, 221–267. [CrossRef]

53. Searle, J.R.; Kiefer, F.; Bierwisch, M. *Speech Act Theory and Pragmatics*; Springer: Dordrecht, The Netherlands, 1980; Volume 10.