Web Intelligence 15 (2017) 19–33
DOI 10.3233/WEB-170350
IOS Press

A modular architecture for intelligent agents in the evented web

J. Fernando Sánchez-Rada *, Carlos A. Iglesias and Miguel Coronado
Grupo de Sistemas Inteligentes, Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, E.T.S.I. Telecomunicación, Avda. Complutense, 30, 20840 Madrid, Spain
E-mails: jfernando@dit.upm.es, cif@dit.upm.es, miguelcb@dit.upm.es

Abstract. The growing popularity of public APIs and technologies such as web hooks is changing online services drastically. It is easier now than ever to interconnect services and access them as a third party. The next logical step is to use intelligent agents to provide a better user experience across services, connecting services with smart automatic behaviors or actions. In other words, it is time to start using agents in the so-called Evented Web. For this to happen, agent platforms need to seamlessly integrate external sources such as web services. As a solution, this paper introduces an event-based architecture for agent systems. This architecture has been designed in accordance with the new tendencies in web programming and with a Linked Data approach. The use of Linked Data and a specific vocabulary for events allows a smarter and more complex use of events. Two use cases have been implemented to illustrate the validity and usefulness of the architecture.

Keywords: Agent architecture, evented web, events, web hooks, Jason

1. Introduction

Our society is relying heavily on online services to store, share and generate new information. As users subscribe to more and more services, the risk of overloading them with notifications becomes evident and problematic [2,19]. When users are frequently interrupted by notifications, their performance degrades and they experience greater anxiety and annoyance [4]. Notifications are not always urgent or important. Some of them require repetitive actions which could be automated. For instance, saving email attachments to your online photo album.

The need for immediacy and interaction between services is leading a new trend in service development, sometimes referred to as *real time web* or *evented web* [39]. The new wave of web services are characterized by their capability to notify each other about new events.

Although this trend made a wide range of sources available, the logic to consume that information is still up to the developer. Intelligent agents are the perfect abstraction for that logic. In particular, personal agents that could use additional personal and contextual information to manage notifications on behalf of users.

Agent systems are developed using existing software agent platforms. These platforms simplify development by eliminating design choices and providing tools (e.g. IDE) and formalisms (e.g. DSL, domain specific language). The main difference between platforms is the agent architecture they implement. Architectures determine how agents are conceptualized, and how agents interact with each other.

The evented web requires an agent architecture that can seamlessly interact with external services. Unfortunately, current agent platforms do not provide any standardized mechanisms to integrate external sources. The integration of sensors and actuators typically requires extending the basic agent architecture and a deep understanding of its implementation.

This paper proposes a new agent architecture for real-time and dynamic scenarios, such as the evented web. This architecture, called Modular Architecture for Intelligent Agents (MAIA), provides an event-

*Corresponding author. E-mail: jfernando@dit.upm.es.

based perspective and a modular design. Additionally, it follows a Linked Data approach to event modeling, which allows agents to reason about these real time scenarios. This paper also explains how this architecture can be embraced for applications that interact with a variable and increasing number of services, as well as its inner workings and implementation challenges. In particular, we describe the use of MAIA to implement a personal cloud agent and a meeting management system.

This paper is structured as follows: Section 2 introduces several concepts of agent platforms, event programming and semantic representation that are relevant to this paper; Section 3 presents an overview of the proposed architecture, and describes its components in detail; Section 4 presents an ontology for describing events; Section 5 describes two use cases for MAIA; Section 6 discusses related work; and in Section 7 we present our conclusions and future work.

2. Background

This section covers three aspects: agent architectures, event-based programming, and semantic representation of events. Section 2.1 gives an overview on agent architectures and platforms, which are the basic pillar for agent development. It also hints the limitations they impose for the integration of evented web services. Event-based programming (Section 2.2) explicitly addresses some of those limitations. For instance, loose coupling and conceptual simplicity ease web development and facilitate integration. Section 2.2 explains what an event-based architecture really is, and some basic concepts behind event-based programming. Lastly, Section 2.3 introduces semantic web concepts which can be leveraged in our agent architecture to provide advanced features.

2.1. Agent architectures

In the 1990s, research interest was focused on the investigation of architectural issues raised by three influential threads of agent research (i.e. reactive agents, deliberative agents and interacting agents), as collected in the excellent survey by Müller [23].

Software agent platforms are usually specialized in a particular agent architecture. For instance, most platforms for deliberative agents have adopted the Belief–Desire–Intention (BDI) model (e.g. Jadex [27], Jack [37] or Jason [8]), while the most popular agent platform for interacting agents, Jade [6], is based on FIPA [36]. Some of these platforms provide facilities to combine reasoning and interacting features, such as Jadex or Jason, which can be integrated with Jade.

The BDI architecture defined by Rao and Georgeff [30] is based on the original model proposed by Bratman for modeling human reasoning [9]. The BDI abstract architecture models human-like reasoning by capturing the mentalistic notions of belief, desire and intention, which are processed according to a generic interpreter. This interpreter assumes that events are atomic and recognized after they have occurred.

Traditionally, both messages and percepts have been managed in the same interpretation cycle, as both are considered forms of external events. Consequently, most agent implementations mix reasoning processes with the communication logic and make them hard to reuse, debug and develop. Recently, several works such as ACRE [22] and Alfonso et al. [1] have proposed to delegate conversation management in a specific module external to the agent reasoning process. The interaction between these two modules is done through actions and perceptions. The reasoning module can reason about the outcomes of every conversation through a set of predefined perceptions, and then execute several actions to manage the status of those conversations (e.g. canceling, forgetting or retrying a conversation).

Several works have proposed different mechanisms for integrating agents and web services, as surveyed in [16]. The existing solutions provide mappings between addressing and messaging schemes in web services and agent systems. They are implemented using a gateway that publishes web service descriptions into FIPA's directory facilitator and vice versa. Nevertheless, this solution is rather complex. For some applications, a more lightweight solution that integrates intelligent agents and web services would be desirable.

2.2. Event-based programming

Event-based programming [25], also called Event-Driven Architecture (EDA) is an architectural style in which one or more components in a software system execute actions in response to one or more notifications. On the web it differs from other approaches in that, instead of the traditional synchronous request-response cycle, the interaction is asynchronous and based on atomic messages. Hence, it is not composed of clients and services but of event producers and consumers.

One of the main advantages of this architecture is that event producers and consumers can be decoupled. Decoupling improves scalability and fault-tolerance. There are three main interaction styles in event programming [25]: *push event distribution*, where event producers emit an event and usually do not expect any specific action by event producers; *channel event distribution*, where event producers send events to an event channel which acts as a broker that redirects the event to event consumers subscribed to that particular event and is usually implemented using Message Oriented Middleware (MOM); and *pull event distribution*, where event consumers follow the traditional request-response pattern to request an event from; an event producers or from an event channel.

Event-based programming has been traditionally popular for programming user interfaces (e.g., Swing or JavaScript) as well as for integration architectures based on a Enterprise Service Bus.

2.3. Semantic representation of events

Ontologies and vocabularies are one of the pillars of the Semantic Web. They are the difference between an ad hoc representation format, and a rich machine-readable format that can be understood, reasoned about, and reused. They contain the definition of concepts and relationships that describe a particular domain. Concepts and relationships are unambiguously represented by a Universal Resource Identifier (URI).

Semantically described resources can be linked to one another, checked for validity, reasoned with, and easily retrieved using SPARQL, the language and API for semantic queries. Furthermore, describing and representing events using semantics allows filtering and selecting events with a high degree of expressibility (including temporal expressions), binding events to external objects they are related to, and creating relations between events in real time (including those that happened in the past or lazy bind object to future events).

There are several ontologies that model the concept of *Event* in different domains and from different perspectives [15,21,28,29]. Among them, the model of events in the Evented Web Ontology (EWE) [14] is the most suitable for the purpose of this work.

EWE stands as a reference model to describe task automations, i.e. rules that command the execution of a particular action when a triggering event occurs. An example of automation may be "when I receive an email, save its attachment in Dropbox".

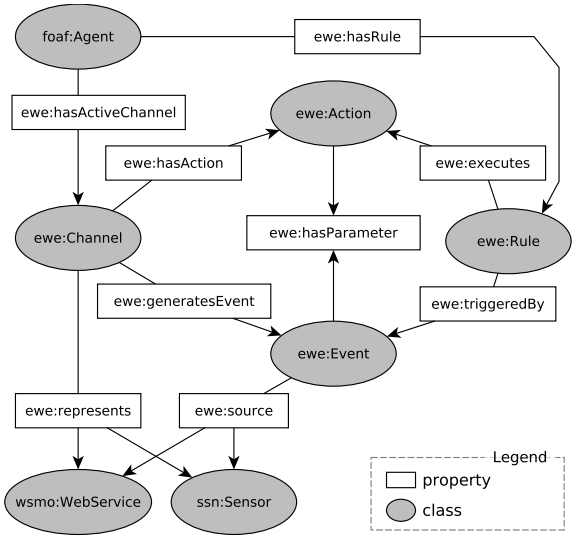

Fig. 1. Detail of EWE Ontology Model.

The core of the ontology comprises four major classes: Event, Action, Channel and Rule (see Fig. 1). An EWE *Event* is an occurrence of a process, e.g. "New email in inbox". An *Action* defines an operation or process provided by an entity, e.g. "Send an email". A *Channel* is any individual that can generate Events, provide Actions, or both. E.g. a mail service, or any other source that generates events, including sensors. *Rule* defines an Event–Condition–Action rule [26], e.g. "send me an SMS when I receive an email if I am not online". The result of executing a rule is an action. Rules involve transferring information from the event to the action that is just created.

As opposed to the definition given in other ontologies [28], EWE defines instantaneous events, i.e. they have no duration over time. The Event class may be subclassed to define particular types of Events. For instance, the class *NewChatMessage* is subclass of Event and defines the type of event that is generated when a new chat messaged is typed. Instances of Event class offer information of the particular event e.g. instances of *NewChatMessage* have information of the chat message and the date when it was sent. We subclass the Event class in Section 5.1 to define an event taxonomy.

In a similar manner to Events, the Action class may be subclassed to specific actions. Again, the definition of an Action is not bound to a Channel, because different channels might provide the same actions. e.g. *Linkedin* and *Facebook* channels provide the *ChangeProfilePicture* action.

3. MAIA architecture

This section discusses the main design choices behind MAIA and presents the main modules of the architecture (Fig. 2), focusing on the relationship between them. Each module is described in greater detail in a separate subsection. The underlying communication mechanism is covered in Section 4.

The driving forces behind the architecture of MAIA are modularity and loose coupling. Every component or functionality is treated as an isolated module. Hence, there are independent modules such as "µblogging", "Calendar", "BDI Platform", etc. The core of the platform then provides the infrastructure to communicate the different modules, plus a set of features to orchestrate and facilitate that communication.

The architecture is designed to allow adding new modules that expand the capabilities of the system. Each module performs a different task (e.g. BDI reasoning, User Interface). All the modules are connected to a core component that controls the flow of information between them. All the communication is carried via events, i.e. atomic messages (Section 4). More specifically, MAIA follows channel event distribution style (Section 2.2).

The main components of the architecture are: the Evented Web Bus, the Agent Bus and the Event Manager. External modules are connected to one of the two buses, either directly or through an adapter (Section 3.1).

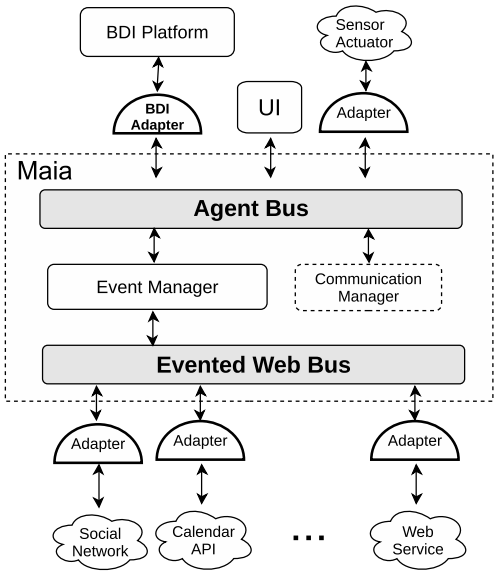

Fig. 2. High level representation of the MAIA architecture.

The Evented Web Bus (Section 3.2) connects to external services. E.g., an email server or a social network. In general, an adapter will be needed.

The Agent Bus (Section 3.2) connects to the modules that are closely related to a typical agent (BDI platform, sensors, actuators, etc.) In particular, all BDI functions and logic are encapsulated in the BDI Platform module, which is connected to the Agent Bus by means of a special adapter (Section 3.1). This platform can be used to develop and run BDI agents that will communicate with the rest of the modules in the architecture.

The Event Manager mediates between both buses, providing extra services to the Agent Bus as described in Section 3.3. These services will have an important role in the development of BDI agents. Section 3.1.2 contains several plans and goals in Agent Speak that make use of these services.

3.1. Adapters

To be able to connect to any of the MAIA buses a module must communicate via events that are MAIA compliant (see Section 4) and use one of the protocols that its bus implements. Unfortunately, not all systems are natively evented. Even when they are, they do not always follow the MAIA events format or use the same protocol as the bus.

An Adapter is a piece of software that mediates between such systems and the rest of the modules. In the best case scenario, which is that of software that is already event oriented, the adaptation process is as simple as translating event formats on the fly and dealing with protocol differences. In the worst case scenario, deeper changes in the software itself might be needed.

We group the adapters in two categories according to the level of integration they provide: basic adapters and Agent Adapters. Basic adapters make the features of an external service or module available to the rest of the modules. Agent Adapters also make the advanced services provided by the Event Manager available to the module in question.

In essence, basic adapters simply add sources of information or interaction with external services, whereas an Agent Adapter connects to a module with more complex logic.

3.1.1. Basic adapters

These adapters take care of: connecting with the Event Manager; translating event formats back and forth; generating MAIA events and storing events for

later consumption. Every adapter that connects to the Evented Web Bus is a basic adapter.

3.1.2. Agent Adapter

Agent Adapters are the interface between an agent system, typically an Agent Platform, and the Agent Bus. The role of these agent systems is to implement the logic of the final application, adding intelligence to the system and communicating to the different modules. The Event Manager provides several services to make it easier to perform certain common actions or simply delegate tasks that would otherwise be done by the agent. Thus, an Agent Adapter should integrate these services in the agent platform.

The design and features of the Agent Adapter highly depend on the target Agent Platform, its internals and the programming interface it offers. Hence, we will focus on the development of an adapter for Jason. Nevertheless, most of the concepts herein are general and apply to other Agent Platforms.

We identified three main challenges in the adaptation process. The first one consisted in communicating with the platform itself, and its individual agents. The second one was translating MAIA events to Jason beliefs. Lastly, there needs to be a way to use the extra services provided by the Event Manager from within any Jason agent. This section covers the first two, whereas Section 5.1 contains excerpts of Agent Speak code to deal with the most common MAIA services.

Every agent within Jason has its own knowledge database, which is populated by data from the different sources. To be able to actually modify the perceptions of the agents, a custom Jason Environment is needed, along with an ad-hoc model for this scenario. By modifying the basic Jason Environment we are able to control not only the sources through which new information is added, but the life cycle of such information.

More precisely, the custom model follows the data inbox concept, the same as regular mailboxes. All information received by the agent is volatile, and will be discarded after it is fetched. Should the agent find the information interesting or necessary for the future, it will save it as beliefs in its permanent knowledge database.

Using these data boxes it is rather easy to integrate our Java code and our agents in AgentSpeak. A special function allows any Java method to send information to any certain agent, and any Java function can be wrapped and made available to the agents in the platform. Figure 3 shows the custom elements created for the adapter.

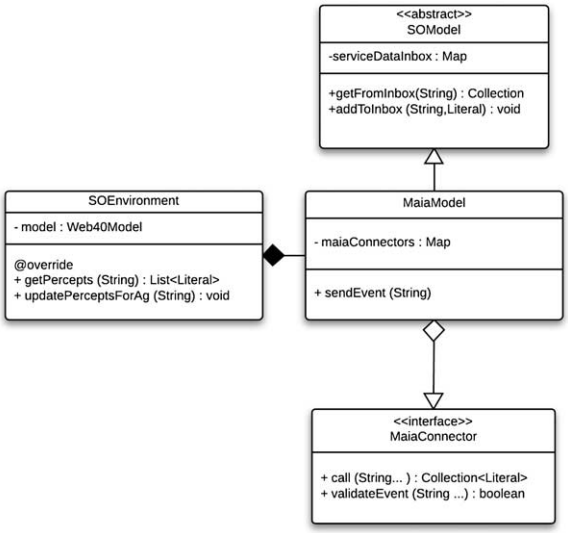

Fig. 3. Adding perceptions to agents in Jason.

Apart from the modifications explained above, events themselves need to be converted to beliefs internally. For this purpose, we created the libraries to translate a subset of the JSON notation to beliefs and vice versa. Unfortunately, the limited syntax of beliefs makes it impossible to perform a complete mapping.

Lastly, it is important to note that every agent should subscribe only to those events that are relevant to its functioning, and to avoid permanently storing them. Otherwise, we risk overloading the agents with too many facts, which hinders the reasoning process and might lead to undesired behaviors.

3.2. The Agent Bus and the Evented Web Bus

The role of the buses is to communicate external nodes with the Event Manager. The architecture differentiates between two buses: the Agent Bus, which connects to the high-level components of the agent, and the Evented Web Bus, to connect to external services. For instance, the Agent Platform, the User-Interface, and the Communication Manager would be connected to the Agent Bus. In contrast, microblogging or web services would interact with the Evented Web Bus.

The reason behind this separation is twofold: it draws a clear line between the access to services and logic, and it allows the Event Manager to provide additional high level services only to the Agent Bus. Most of these services, which will be discussed in the next

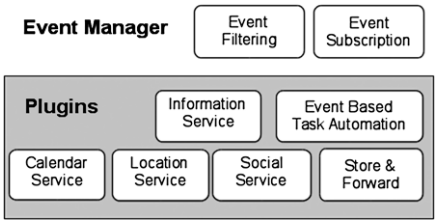

Fig. 4. Internal structure of the Event Manager. High level services are implemented as plugins.

section, are focused on the development of personal agents that interact with social networks.

3.3. Event Manager

The Event Manager (Fig. 4) is the core of the MAIA architecture. It is the bridge between the two buses. One of its roles is to exchange events between them, making Evented Web and sensory information available to agents and forwarding requests from agents to services. However, such information is usually verbose and frequent. Most of the times it is redundant or not critical. In contrast, the communication among agents or between agents and the user interface are usually more critical and sensitive to delays. As a consequence, the exchange between both buses has to be controlled. That is the role of the Event Manager. In addition to plain message passing provided by the buses, the event manager adds event filtering, event subscription, and store and forward. Event Filtering allows selecting only the relevant events in each situation and for each module. By using Event Subscription modules can indicate their interest in certain kind of event which they wish to receive. These subscriptions can be used for event filtering as well. Store and Forward means that modules can receive the events they subscribed to and that were sent while they were disconnected. It also means that events will be saved until they can be forwarded to a module. Without it, an overloaded module would not be able to consume all the events sent to it, which might then be discarded.

Besides controlling the flow of events between different modules, it complements the Agent Bus by providing higher level functions that are not present in it. The Event Manager provides several useful services for the development of personal agents.

Namely, these services are: Identity, Event Based Task Automation, Location, Semantic Information, Social Networks, Calendar and Transactions.

The Identity Service allows agents to define virtual identities. These identities can be linked to the rest of the services. For instance, an identity can be linked to several calendars and social networks. These identities are defined via FOAF [10]. Each identity has a unique ID that can be used to subscribe to the events from the sources linked to it. The Event Based Task Automation offers the option for agents to delegate actions to the Event Manager. These actions will be fired by a certain event, and their result will be another event.

The Social Network service homogenizes the connection and interaction with different social networks. Social networks are an important part of the average user's everyday activity. By integrating them in a personal agent, we can gather relevant information about the user and improve the user's experience. Each social network profile can be linked to several identities. As we saw before, this means the events from different profiles will share a common namespace, making it easy to subscribe to all of them.

The Location service makes it possible to set locations to each identity. Events are sent every time there is a location change, or when a module queries the location of an identity.

The Calendar Service is a common interface to deal with calendars from different sources within MAIA. It is especially meant as an abstraction for online calendar services.

The Information Service offers a simple unified interface for agents to query information from external information sources. As of this writing, the Information service supports SPARQL, being able to send queries to multiple endpoints (DBpedia, data.gov, etc.).

The Transaction service makes it easier for agents to handle operations with online services that follow a known pattern. For instance, the processes between booking a flight and arriving safe to the destination accommodation are quite similar regardless of the flight company, shuttle bus operator, etc. Given that, the Transaction service identifies different events as steps in such processes and acts accordingly to offer extra information to the agents.

4. MAIA events

The communication paradigm in MAIA purposely mimics that of the evented web [39]: all modules communicate through atomic messages called events. An event can be either a notification or a request, in the sense that it may inform of new information (e.g.

"there are 3 new emails") or of an intention to trigger an action in a remote entity (e.g. "send this email").

Events are what confers loose coupling to the architecture. To cover all possible communication needs, the structure and format of these events must cover a wide range of scenarios. Moreover, it is desirable to make events as compatible with the evented web as possible so that the interaction is seamless.

This compatibility must be achieved both on a conceptual level and on a format level.

The conceptual level deals with questions such as: what type of information does an event carry?, how do events relate to each other?, how are modules/services and events related? Most of these questions have already been answered in the previous sections, especially those related to the purpose and usage of events. The Live Web [39] introduces a very generic schema for events. However, a formal definition of the information within events is still missing. The format level relates to how events are represented, serialized and transferred.

EWE (Section 2.3) by Coronado et al. [11] formalizes the idea of events on the web in the form of an ontology. In EWE, there are: Events, which are notifications of new information (e.g. "New email received"); Channels, which are event producers or consumers (e.g. "email service"); and there are Actions, which are the result of the execution of a rule. In MAIA, every module is a Channel. In the case of an adapter, the Channel actually represents the module it is adapting. Since MAIA events (or messages) can be either informative or a request, they are represented with EWE's Events or Actions respectively.

In Listing 1 the semantic representation of an Event and an Action are presented in Notation3 (N3) [7]. They represent a notification about new meeting added to the user's calendar, and the request of sending an email to inform about the meeting. For sake of readability flat instances were used. However, more complex semantic relations may be derided involving third party vocabularies, e.g., to represent the meeting location, its agenda, sender's Persona, etc.

On the other hand, there are several possible formats to serialize semantic information. As shown in the former example, Notation3 is suitable for representing semantic information in a human readable way, however it does not have wide support in most programming languages. To simplify the task of developing new adapters to the evented web, MAIA events use the JSON-LD [35] format in its compact form.

```
@prefix maia: <http://demos.gsi.dit.
    upm.es/maia#>.
@prefix ewe: <http://gsi.dit.upm.es/
    ontologies/ewe/ns#>.

maia:MailChannel a ewe:Channel .
maia:CalendarChannel a ewe:Channel .

maia:NewMeetingAdded_1
    a ewe:Event;
    ewe:source maia:CalendarChannel;
    dcterms:title "new meeting added";
    ewe:title "Meeting name";
    ewe:where "Building A, Room 101";
    ewe:starts "Jul 31, 2015 10:00PM";

maia:SendEmail_1
    a ewe:Action;
    ewe:source maia:MailChannel;
    dcterms:title "send new email";
    ewe:subject "New meeting";
    ewe:recipient "email@example.com";
    ewe:body "A new meeting was
        scheduled".
    ewe:related maia:NewMeetingAdded_1
```

Listing 1. Example of an action and an event in N3.

This approach has multiple advantages: it is a lightweight human-readable format; there are libraries to efficiently process JSON in almost every programming language and JSON-LD libraries have been made for most of them; semantic and non-semantic information can coexist in the same JSON object; and plain JSON information from the evented web might be converted to semantic JSON-LD by adding an appropriate context.

In summary, MAIA events are messages in JSON-LD format that are modeled using the EWE ontology. Events have the following fields:

- **id (@id)** Unique identifier of the sent event for the specified entity (source).
- **timestamp (dcterms:created)** Time of the original emission. This makes time reasoning possible and prevents the side effects of asynchronous communications.
- **source (ewe:source)** Unique identifier of the sending entity.
- **name (ewe:description)** Which describes the event, and is the only required field. Ideally, it will not only consist of a basic string, but of a complete namespace. This allows for a complex

processing of the events and an advanced filtering for triggers. We will get into details later in this section.

- **parameters (ewe:hasParameter)** For any kind of non-trivial event, we will need more information about the entities involved in the event, or the parameters if it is a request. This field is a list of objects that provide further information or parameters. Each object includes the name of the property, and its value.
- **expiration** Used to announce other entities that after this time the success or error callbacks will not be called, to prevent them from replying to or acknowledging the event.

In addition to these fields, a complete JSON-LD object also includes a context, to provide the semantic metadata of each field. Listing 2 contains an event in JSON-LD format. The content of the event is similar to that in Listing 1, with the exception of some additional fields that unambiguously identify event messages.

All events are named following a simple convention, the names are strings separated by double colons, the first string being the name of the module that sent it, for example: *MailChannel::email::new*. Modules use these names to subscribe to events from other sources. For instance, in our previous example a module would need to subscribe to *MailChannel::email::new* to receive the new email events from MailChannel.

What is interesting about MAIA events is that they may contain wildcards * or double wildcards **. Using wildcards, a module can subscribe to a wide range of events. If the name of the event and the name used in the subscription match, the event will be forwarded. A single wildcard replaces/matches any string between double colons (e.g. *a::b::c* and *a::*::c* match). A double wildcard replaces/matches zero or more slots (e.g. *a::b::c* and ***::c* match, and also *a::b::c::**). Wildcards can appear either in the subscription name or in the event name, the comparison is applied symmetrically.

In order to efficiently process these matches and achieve high throughput of events, MAIA buses use an optimized subscription handling algorithm based on subscription trees.

Although one of the aims of the events system is to achieve asynchronous communication, it is worth noting that namespaces and the expiration information allow some sort of remote method invocation. To reply to an event, another event with the name *<source>::success::<id>* or *<source>::error::<id>* can be sent before *Expiration*, where *<source>* is the identifier of the sender and *<id>* is the ID of the original event. These events are currently not being forwarded to the rest of the modules.

As a last comment about the format of events, we have developed adapters for SPARQL and Spotlight endpoints. A W3C recommendation [33] can be used to include the results from SPARQL queries in events.

```
{
 "@context": {
  "ewe": "http://www.gsi.dit.upm.es/
      ontologies/ewe/ns",
  "dcterms": "http://purl.org/dc/terms
      ",
  "id": "@id",
  "@type": "ewe:Event",
  "source": "ewe:source",
  "timestamp": {
   "@id": "dcterms:created",
  },
  "name": "dcterms:title",
  "parameters": {
    "@id": "ewe:hasParameter",
    "@container": "@list",
    "@type": "ewe:Parameter"
  },
  "description": "dcterms:description
      ",
  "title": "dcterms:title",
  "value": "dcterms:value",
 },
 "id": "http://demos.gsi.dit.upm.es/
     maia#MailChannel_"
 "source": "http://demos.gsi.dit.upm.
     es/maia#
     MailChannel_ev_1389937684001
 "timestamp": 1389937684,
 "name": "MailChannel::email::new",
 "parameters": [
   {
     "title": "subject",
     "value": "Testing Maia",
     "description": "Subject of the
         email"
   }
 ],
 "expiration": 1389937694
}
```

Listing 2. Example of an event in MAIA that represents a MailChannel using JSON–LD.

5. Case studies

To clarify some of the concepts explained before, we will present two use cases of the MAIA architecture.

The first scenario consists of a personal agent that uses MAIA to exchange event messages with different web services. This scenario highlights the integration of heterogeneous services with a BDI architecture. Traditional architectures would make this integration much more costly, as it would require modifying the agent cycle. With MAIA, all modifications to the BDI architecture are agnostic of service and only performed once. This is true of other components as well, such as adapters of clients. This high level of modularity is the biggest advantage to MAIA. It is worth noting that this scenario includes both a web client and an Android client, which demonstrate how interaction with the user could take place in such an architecture. These clients could be easily extended and adapted to other use cases just by adding new types of events. Moreover, external services are modeled using the EWE ontology [12], which specifies their events and actions. Thus, intelligent agents can reason and interact with these heterogeneous services in a uniform way, and enables its interlinking with other resources of the Web of Data.

The second scenario covers the implementation of a Task Automation Platform (TAS). The platform connects to physical sensors using an adapter to GSN (Global Sensor Network). Hence, it hints on the use of MAIA beyond its use with Web Services. Such an architecture could be used in smart environments or similar applications.

5.1. Building a personal agent

This section covers the implementation of a personal agent in the travel domain. The aim of this personal agent is to assist users with their trips. This assistance includes: following the process between booking a ticket and arriving to the destination; alerting of any irregularities such as delays, cancellations or forecast alerts; informing users about flight deals during their free days; checking the activity on social networks about topics related to the trip; and handling emails and social activity on behalf of the users when they are away.

For all this to work, the agent will need to connect to: a flight search service; a forecast service; an email server; a calendar service; and a social network. The interaction between the user and the personal agent will be via text messages in natural language. The natural language processing of the messages is delegated to an external REST Natural Language Understanding (NLU) Service. The general architecture presented in Section 3, has been particularized as seen in Fig. 5, so that each of the external services has an associated adapter module.

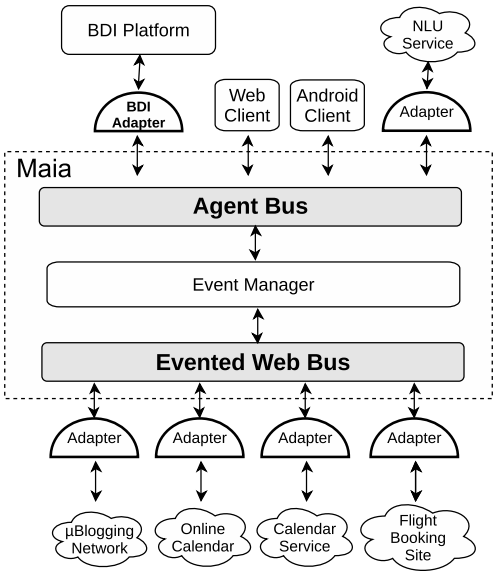

Fig. 5. Architecture of the personal agent prototype.

As mentioned in Section 2.3, events and modules are modeled via Events, Actions and Channels in the EWE ontology. In this case, we formalized the naming convention for each module and the different types of events exchanged in a taxonomy, as shown in Fig. 6. Events are structured in different levels, so that specific Event classes inherit from more general ones. This taxonomy may be used with the event description field of MAIA messages, to subscribe to either particular events, or to events that inherit from another class. For instance, the namespace of events of type *maia:NewTrip* (according to Fig. 6) may be *<Channel>::NewMeeting::NewTrip*. Thus, to subscribe to new messages of that type that are generated by the CalendarChannel we use the namespace *CalendarChannel::NewMeeting::NewTrip::new*. In a similar way, subscriptions to all events of type NewTrip, regardless of the source, may be **::NewTrip::new*; and subscriptions to all events that inherit from NewMeeting may be **::NewMeeting::*::new*.

The logic of the personal agent is provided by a single Jason agent, the travel agent. The agent is running on a modified Jason platform (Section 3.1.2), which

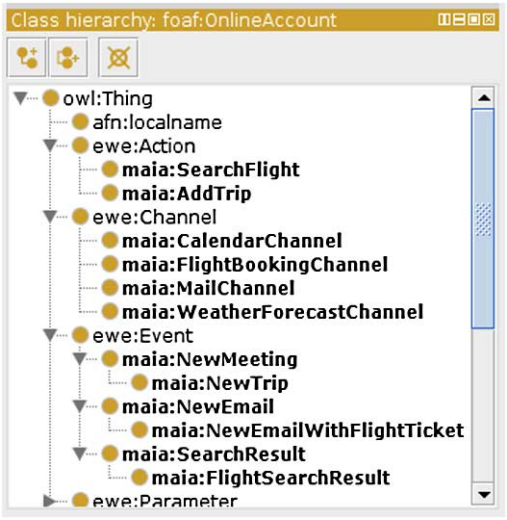

Fig. 6. Taxonomy used to model adapters and messages.

Fig. 7. User interface as a Chrome extension.

is connected to the Agent Bus. The rest of the modules that are connected to the Agent Bus are: a web client that the user can interact with; an Android client, which is analogous to the web client (Fig. 7); and the natural language understanding unit [13], which the agent uses to translate the natural language input into beliefs. Both clients also send the location of the user, so they are both UIs and sensors.

The remaining of this section shows excerpts of code and simplified examples that demonstrate how the higher level services of MAIA, such as location and information services, can be used from within an agent in the Agent Platform. More specifically, it contains AgentSpeak plans to: get the semantic information of the country of the flight destination, which can later be used to fetch more information; alert the user via email when the user has confirmed a flight and the forecast information in the city of origin or destination is negative; subscribe to activity in all the subscribed microblogging sites about the country or city of destination two weeks before the flight, and alert the user about suspicious activity.

The external forecast service can be modeled using the EWE ontology, as shown in Listing 3. Since entities in events are linked to the knowledge graph, agents can use additional information to reason. For example, an agent could access DBpedia [3] to retrieve yearly precipitation from a city and incorporate this information in its plans.

Listing 4 contains a plan to process forecast information during or close to a day of a scheduled flight. To receive such forecast information, the agent must have already subscribed to forecast alerts or any event from the information service.

Listing 5 exemplifies how an agent can query a SPARQL endpoint to get more information. In particular, the query fetches the list of countries, their capitals and their geographic locations if a flight is booked to a city whose country is not known. The query is limited to European cities to use a simple query to a public endpoint (DBpedia).

Lastly, Listing 6 presents a simple example which makes use of the social service. The agent subscribes to microblogging events up to fifteen days before a

```
?event a maia:NewForecast ;
  maia:temperature 16 ;
  maia:city "dbpedia:Madrid" ;
  maia:date "2016-01-01" ;
  maia:forecast "rain" ;
  maia:chances 0.5 .
```

Listing 3. N3 representation of a forecast event.

```
+info("forecast",data(Date,City,
    Temperature,Forecast,Chances))
  : flight(Dept,City,From,To)[id(
      Identity)] | flight(City,Arriv
      ,From,To)[id(Identity)] & ((
      Temperature < 20 | Forecast ==
       "rain" ) & Chances > 0.3 )
  <-!suggest_deals(Identity,Dept,
      Arriv,From,To);
    sendEmail(email_address(Identity
        ),null,"Bad weather for your
         trip",(Date,Temperature,
        Meteo,Chances)).
```

Listing 4. Process forecast information before a flight.

```
+flight(_,City,_,_)
  : ~country(City,_)
  <-info_request("sparql", _, "
    SELECT distinct ?country ?
        capital (SAMPLE(?caplat) AS
         ?caplat) (SAMPLE(?caplong)
         AS ?caplong)
    WHERE {
     ?country rdf:type dbpedia-owl:
         Country .
     ?country dcterms:subject <http
         ://dbpedia.org/resource/
         Category:
         Countries_in_Europe> .
     ?country  dbpedia-owl:capital
         ?capital .
     OPTIONAL {
       ?capital geo:lat ?caplat ;
        geo:long ?caplong . }
    }
    ORDER BY ?country ",country
        (1,2),location(2,3,4,_)).
```

Listing 5. SPARQL query to gather new information.

```
+flight(_,City,Dept(YY,MM,DD,_,_,_),
    _)[id(UserID)]:
  : ((DD > 15 &.date(YY,MM,DD-15)) |
       (.date(YY,MM-1,DD+15))) &
      country(City,Country)
  <-social(event("id",UserID,"social
      ","ublogging","**","stream","
      peak"), [Country,City], ["
      alert","activity","ublogging
      ","away"]).
+event(["alert", "activity", "
    ublogging", _], data(Volume,
    Posts))[id(Identity)]
  : Volume > 10
  <-ui_alert(Identity, "Relevant
      social activity about your
      destination:", Posts).
```

Listing 6. Subscribe to notifications about peaks in activity about the destination of a trip and warn the user via the UI upon alert.

flight is scheduled to depart. The social service will then send alerts about activity when there are enough posts related to the destination city or country. Users are thus informed of noteworthy happenings in the destination country (riots, strikes, concerts, etc.).

5.2. An event-based Task Automation Platform

This case study presents a personal agent for managing work meetings. It aims to assist users to schedule meetings, remind them of upcoming meetings, and support them while the meeting is taking place. The assistance includes setting up the meeting room: unlock the door, switch the lights on, connect the heating system to acclimatize the place, and show the meeting agenda on the wall display.

The particularization of the high level MAIA architecture (Fig. 2) for this implementation is shown in Fig. 8. The agent (BDI Platform) has to connect to the calendar service to know the meeting details e.g. date, agenda, participants. It also needs to connect to the devices (sensors and actuators) in the meeting room to provide in-meeting assistance, e.g. lighting system, door lock, and heating system. The system provides two interfaces that let the agent communicate with the users: with the web client the users can change their preferences about the in-meeting assistance, and the wall display (a screen located in the meeting room) is used by the agent, to show useful information to the participants during the meeting. Finally, the agent also has access to the Event Based Task Automation service, as part of the event manager service library.

In order to be aware of all events from connected sensors or web services, the BDI Agent has to subscribe to events coming from those sources. Web services are connected throughout adapters that imple-

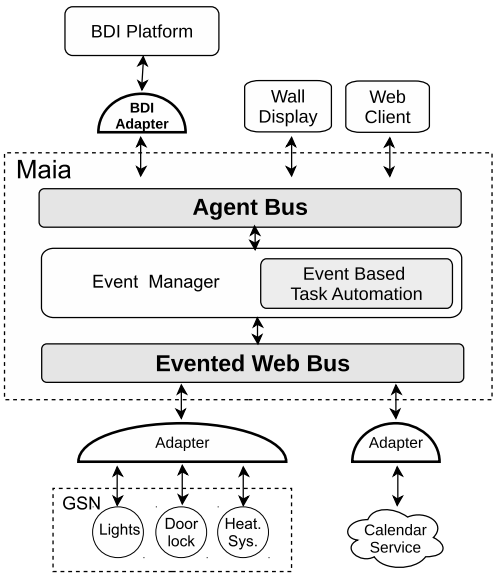

Fig. 8. Architecture of the event-based task automation prototype.

ment their APIs (more elaborated examples of adapters are given in Section 5.1). Sensors and actuators are connected using Global Sensor Network (GSN). GSN is a middleware for deployment and integration of heterogeneous wireless sensor networks. GSN provides *entry points* where event generators (i.e. sensors) push their events to the sensor network, and *accessible exit points* used by the adapter to retrieve events and push them into MAIA. In this case study, we have modified GSN so that it is capable of transport messages to the connected devices (i.e. actuators). In this case, the adapter also uses an entry point to push messages into GSN, so the actuators can retrieve them using the exit point. Hence, the adapter must subscribe to the events that are addressed to the actuators it manages. GSN provides a timestamp for each generated event. It is used to synchronize timestamps from GSN and MAIA so that all events have a common baseline. This allows, MAIA to know the time at which the event happened, instead of the time at which it was pushed to MAIA, removing undesired offsets.

As in the former case study, we model events and event generators as Event/Actions and Channels using the EWE ontology. We define several classes to represent them, as shown in Fig. 9. Events such as *maia:NewMeeting* inform the agent about new meetings according to the information from the web service calendar (*maia:CalendarChannel*). Actions such as *maia:UnlockDoor* or *maia:SwitchLigh* allow the agent so send orders to the actuators connected to GSN. Finally, the wall display is also modeled as a channel (*maia:WallDisplayChannel*), thus the agent can interact with it pushing actions as normal.

The Event Based Task Automation service enables execution of static Event–Condition–Action (ECA) rules [26] without the intervention of the BDI agent. These ECA rules are set by either the agent or the users using the web client. Once loaded in the Task Automation service, ECA rules are fired by events that meets its triggering condition. As a result of a rule execution new events are generated and pushed to MAIA. The Event Based Task Automation service is used to define rules such as "If the temperature of the meeting room is below 23C, switch the heating system on". The former rule is fired by event messages from the temperature sensor with a measurement under 23. The resulting event represents an intent to switch the heating system on. The adapter of the GSN will receive this event and instruct the heating system to turn itself on.

Since events are represented using EWE, for convenience we use SPARQL for describing rules as recommended by [14]. Thus, in the prototype, the Task Automation service implements a SPARQL engine (as well as a rule repository to store the loaded rules). Continuing with the former example, the rule in its SPARQL form is presented in Listing 7.

As mentioned, shifting rules to the Task Automation service frees the agent from executing these rules itself. Nevertheless, the agent is responsible of managing them, e.g. watching from collisions between rules set by users, remove deprecated rules, introduce additional rules. Listing 8 shows the AgentSpeak goal that

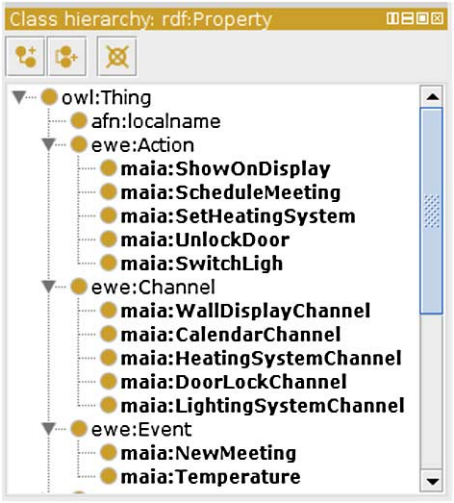

Fig. 9. Taxonomy used to model adapters and messages.

```
CONSTRUCT{
  ?action a maia:SetHeatingSystem ;
    maia:setTemperature 26 .
}
WHERE {
  ?event a maia:Temperature ;
    maia:temperature ?temp .
}
FILTER ( ?temp < 23 )
```

Listing 7. SPARQL representation of the rule "If the temperature of the meeting room is below 23C switch the heating system on".

```
+has_ended(eventID)
  : ?ruleforevent(rule, eventID)
  <- removeRule(rule)
```

Listing 8. Rule for managing Task Automation service.

removes all rules from the Task Automation service that are related to a concluded meeting.

6. Related work

Several authors have addressed the definition of an event based agent architecture. Munteanu [24] proposes an event-based middleware for Cloud Governance based on multiagent system. Their work is focused on identifying the agent roles for cloud governance and does not deal with engineering an event-based agent system. Thus, our solution can complement their proposal since it provides a suitable architecture for event-based processing.

In the first prototypes of this system, different multi agent system platforms were evaluated. The most promising of them is SPADE (Smart Python multi-Agent Development Environment) [17]. SPADE is based on the XMPP messaging and presence protocol. The resulting architecture is thus similar to MAIA: a series of nodes connected to a central XMPP server, with a special module to handle communication and FIPA features. The XMPP protocol provides many of communication features: publish-subscribe mechanism to allow push updates, form-data to manage work-flow between user, libraries for many programming languages and platforms, etc. However, SPADE requires a modified XMPP server, and the protocol itself is quite complex, making the use of external libraries mandatory. In addition to this, agent communication is tied to the FIPA standard. In contrast, MAIA offers a simpler and lighter protocol, which makes it more convenient to develop new applications for the evented web.

MAIA includes the basic elements and protocol to exchange information between different parties. However, agent communication is a more sophisticated process that has been treated broadly in other texts [22]. There are very complex agent communication solutions.

Although MAIA focuses on a different problem, and aims to be simple to implement, it was also designed to cover those cases. Instead of covering every possible scenario, MAIA can be extended by adding new functionality to the Event Manager. In particular, to achieve a scheme similar to Lillis et al. [22], two additions would be needed: one in the agent level, adding the communication logic and protocols; and another one on the platform level, to allow agents to announce or subscribe their services, share protocol definitions or act as a mediator in disputes. The first addition would be made on top or within the MAIA adapter, if it is not already contemplated in the agent platform. The second one is labeled as Communication Manager module in the MAIA architecture. This paper will not cover this specific module, but it is important to note that the architecture was created with it in mind.

Other authors have proposed architectures for agents in events environments such as Internet of Things (IoT) [5,20].

CALM [20] proposes the use of a Multi-Agent Systems (MAS) communication platform on top of a standard MOM based on CORBA to provide contextual services. This MOM is accessed both by devices and by agents and uses COIDL based on omniIDL [18]. Our proposal differs from CALM in several aspects. Firstly, we propose a design pattern for architecturing event-based agent systems, where two MOM are used for managing external and internal events. Secondly, we propose a linked data approach for representing events, and adapters in charge of transforming external data to this semantic event format. Thus, we can compare only the integration of external events in Maia and CALM. Our proposal is compatible with external services as well as accessing linked data end points and thanks to the usage of JSON-LD it is independent of its implementation, while CALM relies in CORBA technology.

Barbero et al. [5] propose an IoT platform based on a Service Oriented Architecture (SOA) architecture. Their architecture is based on two main components: a web services management that interacts with external services and a contextual awareness module that process this information. The contextual awareness module consists of semantic reasoner modules and processing agents, that can subscribe to certain events. An ontology is defined for entities and contextual services. Processing agents are domain specific, and are in charge of filtering events, dealing with sensor data and provide contextual services. The main differences from our proposal is that we encapsulate event processing in the Event Manager, while Barbero et al. distribute this function across the semantic reasoner and the processing agents. In addition, our proposal is able to integrate existing agent platforms such as Jason.

7. Conclusions and future work

The architecture presented in this paper proves that it is possible to achieve modern systems that combine

the potential of intelligent agent systems and the interconnection and ever-growing applications of the modern web.

The resulting application goes beyond the state of the art, putting together already existing solutions from different fields. It thus shows that we can make good use of the existing technologies to implement innovative ideas.

In this paper, the most important shift is in the way we understand agents and agent communication. Hence, MAIA focuses on describing how an agent architecture can be conceived as a set of cooperating modules that communicate each other through a bus. As mentioned in the introduction, one of the functionalities that can benefit of this approach is the decoupling of reasoning and agent communication. The design and integration in MAIA of this communication module [1] as well as interaction protocols [34] such as negotiation [40] deserves more attention and is left as future work.

One of the main aspects to improve from a pragmatic point of view is the security of the information exchanged and the scope in which it is visible. Currently MAIA allows username/password authentication and mechanisms to control event subscription on a per-module basis. Other security measures and mechanisms such as anonymity [38] would be of interest in e-commerce and auction applications.

Another field for future research is to further expand the definition of events to include other concepts such as propagation of events. This might lead to delegation and collective planning, but it also poses challenges related to agent communication.

Finally, we are working on improving the interaction with users by incorporating affects as well as multimodal interaction. MAIA's Linked Data approach together with a Linked Data multimodal representation of emotions [31,32].

Acknowledgements

This work was supported by the European Union through the SMARTOPENDATA FP7 Project (Grant Agreement no: 603824), by the Spanish Ministry of Economy and Competitiveness under the R&D project SEMOLA (TEC2015-68284-R) and by the Autonomous Region of Madrid through programme MOSI-AGIL-CM (grant P2013/ICE-3019, co-funded by EU Structural Funds FSE and FEDER). The authors would like to thank Carlos Crespo for his work in the implementation of the prototype described in Section 5.2 as part of his master thesis.

References

[1] B. Alfonso, E. Vivancos, V. Botti and A. García-Fornes, Integrating Jason in a multi-agent platform with support for interaction protocols, in: *Proceedings of the SPLASH '11 Workshops*, SPLASH '11 Workshops, ACM, New York, NY, USA, 2011, pp. 221–226.

[2] L. Ardissono, G. Bosio and M. Segnan, An activity awareness visualization approach supporting context resumption in collaboration environments, in: *International Workshop on Adaptive Support for Team Collaboration*, 2011, pp. 15–25.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak and Z. Ives, *DBpedia: A Nucleus for a Web of Open Data*, Springer, 2007.

[4] B.P. Bailey, J.A. Konstan and J.V. Carlis, The effects of interruptions on task performance, annoyance, and anxiety in the user interface, in: *Proceedings of INTERACT*, Vol. 1, IOS Press, 2001, pp. 593–601.

[5] C. Barbero, P.D. Zovo and B. Gobbi, A flexible context aware reasoning approach for IoT applications, in: *2011 12th IEEE International Conference on Mobile Data Management (MDM)*, Vol. 1, 2011, pp. 266–275. doi:10.1109/MDM.2011.55.

[6] F.L. Bellifemine, G. Caire and D. Greenwood, *Developing Multi-Agent Systems with JADE*, Wiley Series in Agent Technology, John Wiley & Sons, 2007.

[7] T. Berners-Lee, Notation 3 – An readable language for data on the web, 2006.

[8] R.H. Bordini and J.F. Hübner, BDI agent programming in AgentSpeak using Jason, in: *Proceedings of 6th International Workshop on Computational Logic in Multi-Agent Systems*, LNCS, Vol. 3900, Springer, 2005, pp. 143–164.

[9] M.E. Bratman, *Intention, Plans, and Practical Reason*, Harvard UP, Cambridge, Mass., 1987.

[10] D. Brickley and L. Miller, FOAF vocabulary specification, 2014, available at http://xmlns.com/foaf/spec/.

[11] M. Coronado and C.A. Iglesias, EWE ontology: Modeling rules for automating the evented web, 2013, available at http://www.gsi.dit.upm.es/ontologies/ewe/.

[12] M. Coronado and C.A. Iglesias, Task automation services: Automation for the masses, *Internet Computing, IEEE* **20**(1) (2016), 52–58. doi:10.1109/MIC.2015.73.

[13] M. Coronado, C.A. Iglesias and A.M. Mardomingo, A personal agents hybrid architecture for question answering featuring social dialog, in: *2015 International Symposium on INnovations in Intelligent SysTems and Applications*, 2015.

[14] M. Coronado, C.A. Iglesias and E. Serrano, Modelling rules for automating the Evented WEb by semantic technologies, *Expert Systems with Applications* **42**(21) (2015), 7979–7990. doi:10.1016/j.eswa.2015.06.031.

[15] A.R.J. Francois, R. Nevatia, J. Hobbs, R.C. Bolles and J.R. Smith, VERL: An ontology framework for representing and annotating video events, *MultiMedia, IEEE* **12**(4) (2005), 76–86. doi:10.1109/MMUL.2005.87.

[16] D. Greenwood, M. Lyell, A. Mallya and H. Suguri, The IEEE FIPA approach to integrating software agents and web ser-

vices, in: *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '07*, ACM, New York, NY, USA, 2007, pp. 276:1–276:7.

[17] M.E. Gregori, J.P. Cámara and G.A. Bada, A Jabber-based multi-agent system platform, in: *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, ACM, 2006, pp. 1282–1284.

[18] D. Grisby, S.-L. Lo and D. Riddoch, *The Omniorb Version 4.1 User's Guide*, Apasphere Ltd. and AT&T Laboratories Cambridge, 2009.

[19] T. Gross, W. Wirsam and W. Graether, Awarenessmaps: Visualizing awareness in shared workspaces, in: *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, ACM, 2003, pp. 784–785. doi:10.1145/765891.765990.

[20] S. Han, S.K. Song and H.Y. Youn, CALM: An intelligent agent-based middleware for community computing, in: *Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance, SEUS 2006/WCCIA 2006, The Fourth IEEE Workshop on*, 2006, p. 6.

[21] T. Kushida, T. Takagi and K.I. Fukuda, Event ontology: A pathway-centric ontology for biological processes, in: *Pacific Symposium on Biocomputing*, 2006, pp. 152–163.

[22] D. Lillis, Internalising Interaction Protocols as First-Class Programming Elements in Multi Agent Systems, PhD thesis, University College Dublin, 2012.

[23] J.P. Müller, Architectures and applications of intelligent agents: A survey, *The Knowledge Engineering Review* **13**(4) (1999), 353–380. doi:10.1017/S0269888998004020.

[24] V.I. Munteanu, T.-F. Fortis and V. Negru, An event driven multi-agent architecture for enabling cloud governance, in: *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, UCC '12*, IEEE Computer Society, Washington, DC, USA, 2012, pp. 309–314.

[25] E. Opher and P. Niblett, *Event Processing in Action*, Manning Publications Co., 2010.

[26] G. Papamarkos, A. Poulovassilis and P.T. Wood, Event–condition–action rule languages for the semantic web, in: *SWDB*, Citeseer, 2003, pp. 309–327.

[27] A. Pokahr and L. Braubach, From a research to an industry-strength agent platform: Jadex v2, in: *Business Services: Konzepte, Technologien, Anwendungen. 9. Internationale Tagung Wirtschaftsinformatik*, 2009, pp. 769–780.

[28] Y. Raimond and S. Abdallah, The event ontology, Technical report, 2007, available at http://motools.sourceforge.net/event.

[29] Y. Raimond, S.A. Abdallah, M.B. Sandler and F. Giasson, The music ontology, in: *ISMIR*, Citeseer, 2007, pp. 417–422.

[30] A.S. Rao, M.P. Georgeff et al., BDI agents: From theory to practice, in: *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, San Francisco, 1995, pp. 312–319.

[31] J.F Sánchez-Rada, C.A. Iglesias and R. Gil, A linked data model for multimodal sentiment and emotion analysis, in: *Proceedings of the 4th Workshop on Linked Data in Linguistics: Resources and Applications*, Beijing, China, Association for Computational Linguistics, 2015, pp. 11–19.

[32] J.F. Sánchez-Rada and C.A. Iglesias, Onyx: A linked data approach to emotion representation, *Information Processing & Management* **52** (2016), 99–114. doi:10.1016/j.ipm.2015.03.007.

[33] A. Seaborne, SPARQL results in JSON, 2011, available at http://www.w3.org/TR/sparql11-results-json/.

[34] J.M. Serrano and S. Ossowski, A compositional framework for the specification of interaction protocols in multiagent organizations, *Web Intelligence and Agent Systems: An international Journal* **5**(2) (2007), 197–214.

[35] M. Sporny et al., JSON-LD 1.0, 2014, available at http://json-ld.org/spec/latest/json-ld/.

[36] D. Steiner, FIPA: Foundation for intelligent physical agents – Das aktuelle schlagwort, *KI* **12**(3) (1998), 38.

[37] P. Wallis, R. Ronnquist, D. Jarvis and A. Lucas, The automated wingman – Using JACK intelligent agents for unmanned autonomous vehicles, in: *Aerospace Conference Proceedings*, Vol. 5, IEEE, 2002, pp. 2615–2622.

[38] M. Warnier and F. Brazier, Anonymity services for multi-agent systems, *Web Intelligence and Agent Systems: An International Journal* **8**(2) (2010), 219–232.

[39] P.J. Windley, *The Live Web: Building Event-Based Connections in the Cloud*, Course Technology, 2011.

[40] X. Zhang, V. Lesser and T. Wagner, A layered approach to complex negotiations, *Web Intelligence and Agent Systems: An International Journal* **2**(2) (2004), 91–104.