Service-Oriented Middleware for the Cooperation of Smart Objects and Web Services

Andrea Giordano and Giandomenico Spezzano

Abstract Many physical devices can be interconnected and cooperate by Internet of Things (IoT), providing and consuming information available on the network. These will not only provide information by monitoring the real-world, but create complex collaborations, interacting also with business processes, in order to provide sophisticated value-added services. In addition, business processes can also adapt their behavior in response to real-time context updates. Web services technology offers a promising approach to provide information and functionalities of physical objects to business processes, since it facilitates interoperability and encapsulates the heterogeneity and specificity of physical objects. To address the dynamic composition of web services in a decentralized, distributed manner, with no single point of failure, a choreography execution model can be used. This chapter describes an approach to support adaptable business processes (workflows) considering changes in the state of Things; likewise, whenever needed, the software controlling the behavior of sensors can be dynamically configured as a result of changes in the functional specifications of business processes.

1 Introduction

From an enterprise and economic perspective, the Future Internet will be the basis for *a web-based service economy* [1] that will merge the digital and physical worlds. The Future Internet, it is now widely accepted, will have four pillars [2]. Besides the Internet of Networks, there will be an Internet of Services as well as Internet of Things integrating common objects into our lives. Finally, an Internet of Contents &

A. Giordano · G. Spezzano (✉)
CNR-ICAR, Via P. Bucci 41C, 87036 Rende, CS, Italy
e-mail: giordano@icar.cnr.it

G. Spezzano
e-mail: spezzano@icar.cnr.it

G. Fortino and P. Trunfio (eds.), *Internet of Things Based on Smart Objects*,
Internet of Things, DOI: 10.1007/978-3-319-00491-4_3,
© Springer International Publishing Switzerland 2014

Knowledge, and an Internet of People are foreseen too. It is important to note that these terms should not be regarded as different "Internets" that will exist in parallel, but rather as different aspects of a common Future Internet.

The innovative and rapidly evolving area of Internet of Things [3] and Services (IoTS) integrates two of the four pillars of the Future Internet and investigates a world where smart objects (SOs) [4–7]—that is, autonomous physical/digital objects augmented with sensing, processing, and network capabilities, are seamlessly integrated into the information system where they can become active participants in business processes. The supporting service-oriented middleware [8] will then abstract the functionalities of SOs as services as well as provide the needed interoperability and flexibility, through a loose coupling of components and composition of services.

Efforts in this area are focused on a development of platforms and solutions where services and SOs cooperate and can be employed in real-world applications in industrial domains such as manufacturing, e-Health, smart cities, home automation, e-Business, etc. However, owing to the heterogeneity of devices and tight coupling of individual information systems, developers cannot easily create their specific applications by combining physical devices and web resources. To address these problems, we proposed to realize composite applications combining services and SOs by event-driven choreographic workflows.

Nowadays enterprise systems are built on a service-oriented architecture (SOA), and business processes in such systems are modeled as an orchestration of underlying services. In order to integrate the IoT into business process systems it is necessary to service-enable IoT resources, e.g., the sensors and actuators that are used to interact with the physical environments. The current state-of-the-art is mostly focused on integration of IP enabled networked smart objects where nodes communicate their information using RESTful Web services. We argue that the approach for the integration of RESTful SOs with existing, widely deployed SOA technologies such as Web services and Business Process Execution Language (BPEL) is the key to the success of SOs in enterprise systems.

In SOA, service composition is normally achieved either through a centralized controller or in a decentralized manner. Support of decentralized workflow execution and scalability are important issues for workflow management systems since it makes it easier to obtain a flexible and adaptive composition of services.

Typically, to reach the required level of scalability, the workflow management system must be distributed and make use of replicated web services that are selected and used at runtime. Traditional workflow systems use centralized orchestration techniques which limit the scalability in the presence of a high number of services. In the orchestration model, all data pass through a centralized engine, which results in unnecessary data transfer and wasted bandwidth so that the engine becomes a bottleneck to the execution of a workflow. Choreography techniques [9], although more complex to model, offer a decentralized alternative and are optimal architectures for data-centric workflows. In this model, data are passed directly to where they are required, at the next service in the workflow.

Self-organizing in this context describes the adaptability of the model during deployment. Changes in the environment (e.g. location change, connectivity outage,

reconfiguration of business processes etc.) require reorganization of the deployed components during run time to meet given Quality of Service (QoS) constraints.

Our idea is to integrate enterprise web services with RESTful SOs by exploiting the concept of service *choreography* undertaking the scalability and dynamicity issues of IoT in order to extend the existing (adaptable) service composition mechanisms. We show that applications involving SO interaction can be seen as a particular case of event-driven composite services.

To this end, the rest of chapter is structured as follows: in Sect. 2, we present our view of system architecture for the execution of event-driven workflows; Sect. 3 presents a description of the adaptive P2P agent-based framework, called Sunflower, we studied and designed, that supports autonomic management of workflows; Sect. 4 describes the integration in Sunflower of RESTful SO; Sect. 5 illustrates the details of the proposed methodology through a case study; finally Sect. 6 concludes the chapter.

2 System Architecture

This section presents an architecture for the execution of event-driven workflows (i.e. composite applications combining services and SOs through events). The inner part of Fig. 1 shows the architecture of the Sunflower service execution platform designed to support the composition of SOAP services. This cooperating model is created by WS-BPEL workflows. Sunflower permits a decentralized and optimized execution of WS-BPEL workflows upon a P2P system as described in Sect. 3.

In this context, addressing Smart Object (SO) technology and, in general, Internet of Things philosophy requires some additional mechanisms to suitably cope with physical entities. Firstly, a kind of transport layer should be chosen and implemented in order to foster proper interaction between the WS-BPEL world and concrete “things” (i.e. SO). Secondly, given that SOs capture the state of the environment in which they are embedded, environmental state modifications should be carefully handled and reflected at the Sunflower side. Finally, a mechanism is required that permits Sunflower workflows to trigger actuation upon SOs.

On the basis of the previous considerations, we propose the use of Web Service Proxy (SP) acting as an adapter/wrapper of the SO’s world. Through SP, each command coming from Sunflower will be forwarded toward the SOs. In addition, each environmental state modification will be considered as an event and notified to the Sunflower part. In summary, the proposed system can be seen as a Web Service orchestrator enhanced by a SOs mashup and an even-driven engine.

SP adds the following features to Sunflower:

- support for a combination of services implemented by means of different technologies (e.g. SOAP, REST etc.);

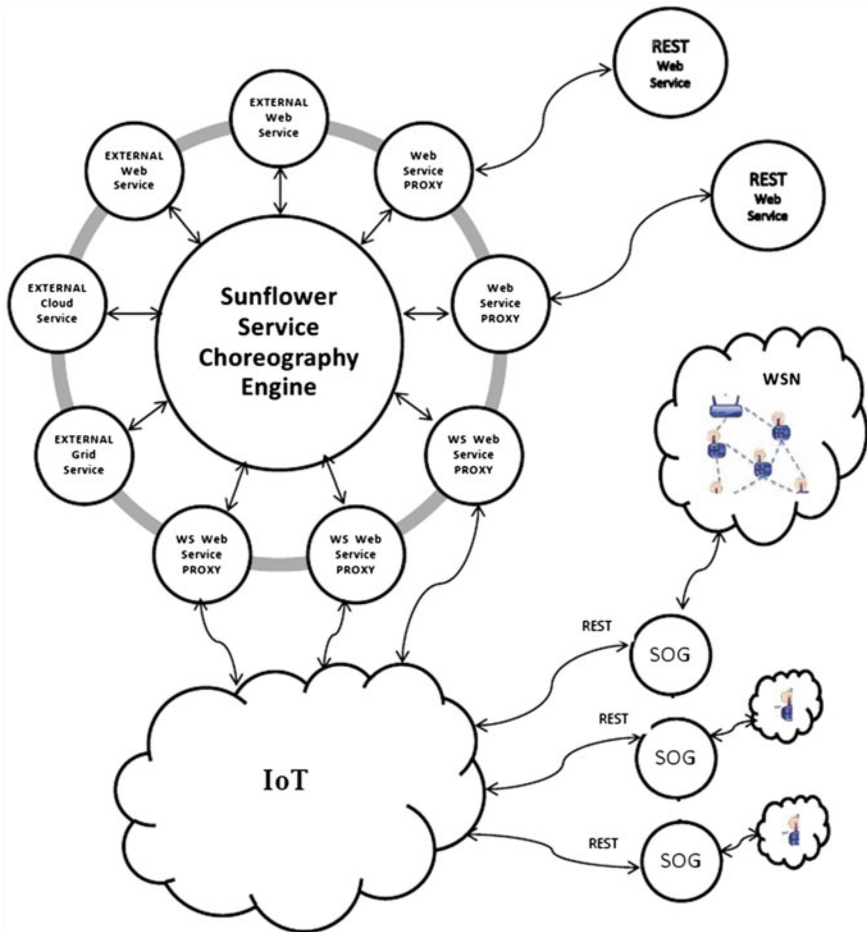

Fig. 1 Architecture for event-driven choreographic workflows execution combining Web services and RESTful Smart Objects

- interaction between Web service and SOs through an "event/action" paradigm, according to which, the occurrence of an event triggers the execution of one or more actions in other components.

SP is attached to SOs via REST invocations on a middleware layer that is in charge to manage underlying SOs. This middleware layer is represented by a so-called Smart Object Gateway (SOG). SOG offers a transparent and ubiquitous access to the physical part due to a well-established interface exposed as a REST service as described in Sect. 4.

SOG allows enterprise application to connect directly to devices without using proprietary drivers or addressing some kind of fine-grained technological issues.

In addition, it fosters the reusing of a pre-existent Web Service in conjunction with SOs thus achieving a perfect match between the Internet of Service and Internet of Things.

The low level of our architecture concerns formalization of SO and how it is integrated in the system. We define a SO as a system made up of one or more physical devices that together achieve complex behaviors. Each SO comprises "functionalities" directly provided by the physical part.

Essentially, a SO exposes an abstract representation (i.e. *machine-readable descriptions*) of the features and capabilities of physical objects spread in the smart environment. It is implemented as computer software that is used to link physical objects with the virtual world.

Functionalities exposed by different types of SOs can be combined in a more sophisticated way on the basis of event-driven rules which affect high-level applications and end-users. A SO is self-managed and self-configurable, capable of being used also out of the context for which it was initially created.

Each physical object, contained in a SO, should automatically perform a *simple* action (e.g., lighting, recording) in response to a *simple* event (e.g. detecting a user, people who sit in a chair). On the other hand, SOs must have the flexibility to change their behavior dynamically on the basis of complex applications even though they posses low processing power and small memory.

A SO changes the behaviour of physical objects by remote and dynamic reprogramming thus considering them as execution parts of business processes.

In summary, our architecture is structured in three layers:

- The first level is the SO level dedicated to the characterization of the SO abstraction in terms of its different *functionalities* that can be either sensing or actuating and can be refined by further parameters that dynamically configure it (see Sect. 4.1).
- The second level is dedicated to SOG abstraction which permits operations of remote and transparent reading and writing on SOs and, also, definition of complex rules on SO functionalities.
- The top level concerns applications written as WS-BPEL/Sunflower workflows. This level encapsulates SOs through Web Service Proxies linked via REST to the underlying layer in order to hide the heterogeneity of devices and provide a seamless way to integrate SOs with web applications.

3 Sunflower Framework

Sunflower is an adaptive P2P agent-based framework for configuring, enacting, managing and adapting autonomic workflows [10, 11]. Sunflower assumes that multiple copies of a Web service co-exist, with different performance profiles and distributed in different locations. During the execution of the workflow, when a service fails or becomes overloaded, a self-reconfiguring mechanism based on a *binding adaptation* model is used to ensure that the running workflow is not interrupted but its

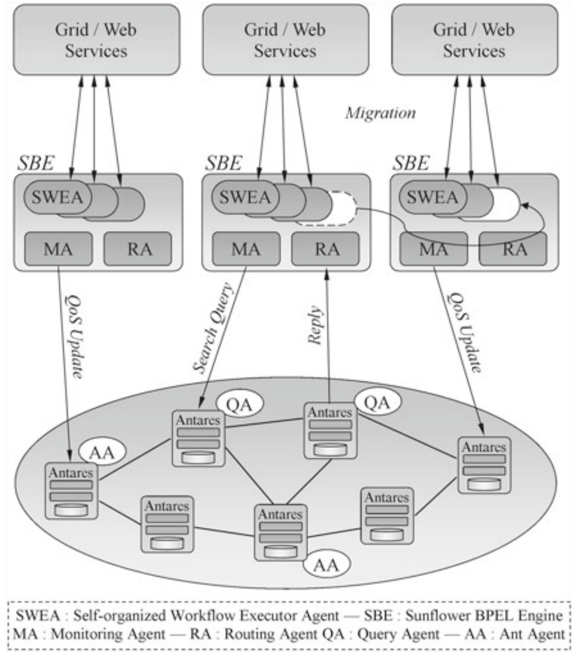

Fig. 2 The Sunflower architecture

structure is adapted in response to both internal or external changes. Figure 2 shows the architecture of the framework Sunflower.

Workflows are described in Sunflower by the BPEL language [12] in order to exploit existing design tools. Sunflower replaces the standard BPEL engine with a new decentralized engine able to exploit the dynamic information available in the network and respond to the dynamic nature of Internet.

The workflow process is enacted by a set of cooperating Sunflower BPEL engines (SBE), instantiated at all participating nodes, which are responsible for interpreting and activating part of the process definition and interacting with the external resources—*invoked web services*—necessary to process the various activities.

A dynamic group of bio-inspired mobile agents *SWEA* (Self-organized Workflow Executor Agent) [13] representing the workflow executors generated from the BPEL workflow specification are initially deployed on the basis of the workflow configuration. A coordination model describes how the generated agents cooperate with each other to reach a choreographic execution. In Sunflower, the coordination model is obtained by the Petri Net (PN) associated with the BPEL program. The PN representation is then structurally decomposed into a set of distributed sub-flow schemas.

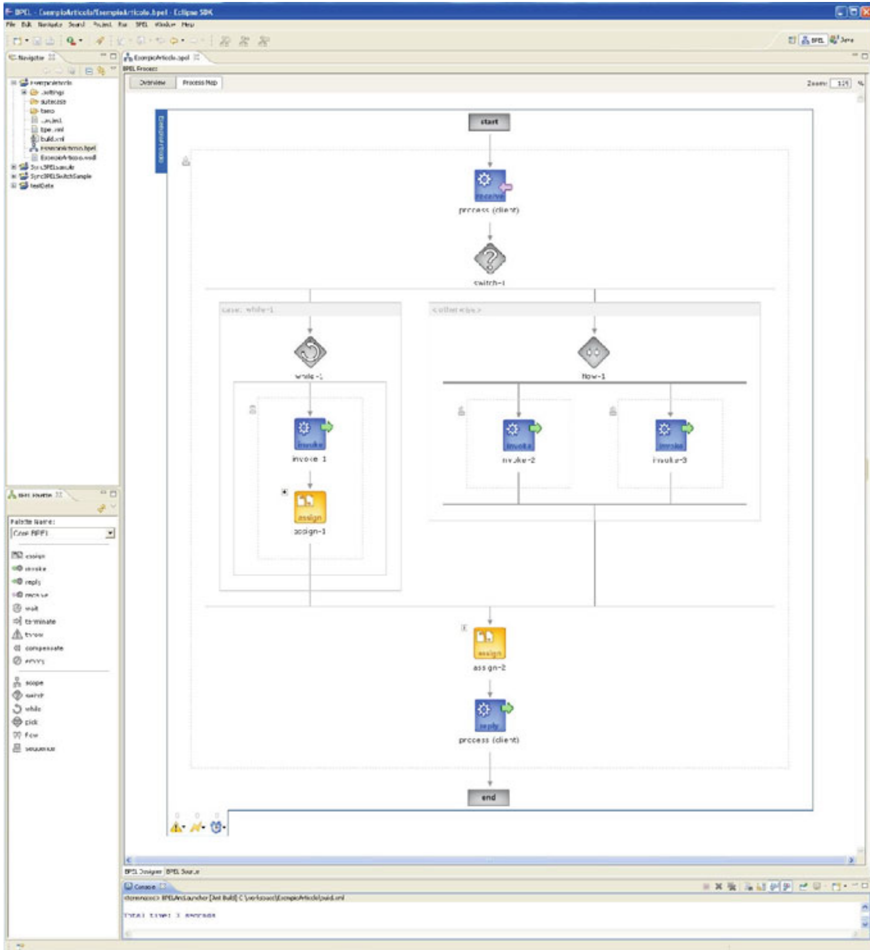

Fig. 3 Example of BPEL workflow (*right*)

3.1 Mapping BPEL Workflows on Petri Nets

A workflow described in BPEL, as shown in Fig. 3, details the flow control and any data dependencies among a collection of Web services being composed. We build every process in a BPEL workflow by plugging language constructs together; we thus can translate each construct of the language into a Petri net (PN). Each primitive or structured activity can be easily modeled as a Petri Net as illustrated in Fig. 4.

BPEL based workflows are converted to a Petri net applying the rules defined by the Van der Aalst methodology [14] that generate a PNs via the repetitive replacement of elemental PNs with other PNs. Figure 5a shows the conversion of the BPEL workflow described in Fig. 3 to a PN form using the replacement property.

BPEL construct	Petri net
<sequence> activity activity ... activity </sequence>	<sequence> activity activity activity <sequence>
<flow> <sequence/> <sequence/> ... <sequence/> </flow>	<flow> <sequence> <sequence> <sequence> </flow>
<switch> <case/> ... <case/> <otherwise/> </switch>	<switch> <case> <case> <otherwise> </switch>
<case condition > activity activity ... activity </case>	<case> condition activity activity activity <case>
<otherwise> activity activity ... activity </otherwise>	<otherwise> activity activity activity <otherwise>
<while condition > activity </while>	<while> activity condition </while>
<invoke> or <assign> or <receive> or <reply>	[<assign>|<invoke>|<receive>|<reply>]

Fig. 4 Example of BPEL constructs (*left*) converted to Petri nets (*right*)

3.2 Petri Nets Partitioning

A workflow written as a single BPEL program must be decomposed in an equivalent set of decentralized processes to set up the choreography model. Our strategy is to construct a PN for the workflow and then apply partitioning rules to operate on such an abstract representation to create the set of cooperating sub-workflows. Our PN partitioning algorithm is based on the idea of merging tasks starting from the *invoke* activities along the control dependence edges.

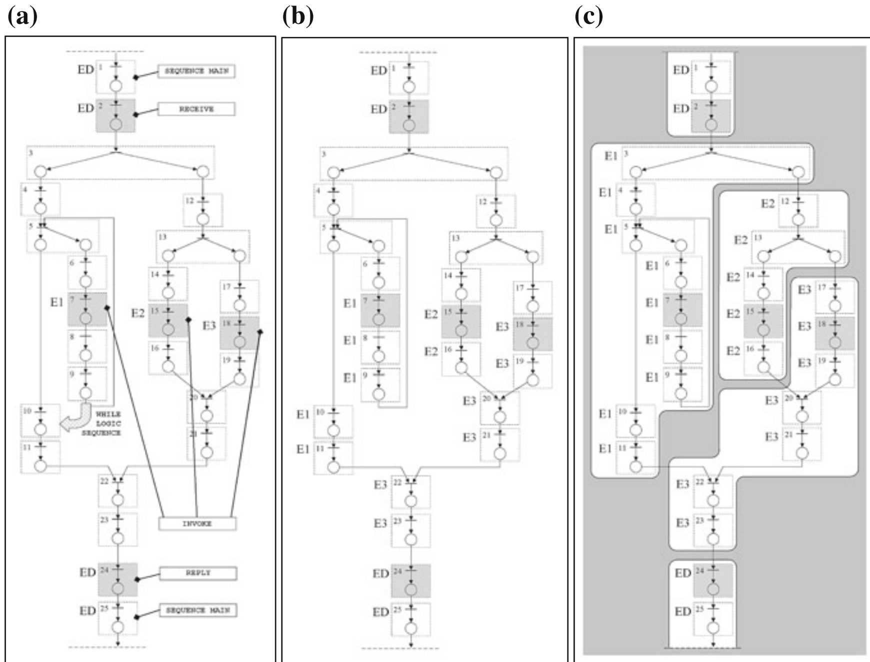

Fig. 5 Petri nets partitioning **a** Petri nets. **b** Top down allocation. **c** Bottom up allocation

An informal description of the partitioning algorithm is as follow:

- The begin and end portions of PN concerning the *main sequence* must be assigned to the same peer, named Peer Collector (PC). *Reply* and *receive* activities must be also executed on the PC.
- The portion of PN concerning the *invoke* is assigned to the peer handling the web service called by the invoke itself.
- All the other constructs are assigned by means of two subsequent visits (*top down* and *bottom up*) of the PN graph. The visits of the PN graph start from the invoke activities. Constructs between two invoke activities can be assigned to one of the two peers where the invokes are executed in order to balance the load.

After the initial allocations, we start with an TopDown visit of the PN graph and then continue with a BottomUp visit. The procedure is as follow:

1. *TopDown initial allocation*: for each PN portion concerning the invoke activity, the label that indicates the peer on which the invoke activity is allocated is propagated to all the *successors*; in the case of controversy (several activities going to the same place), only the *right* label is propagated.

2. *BottomUp final allocation*: for each PN portion concerning the invoke activity, the label of the peer assigned to this activity is propagated to all the *predecessors*; in the case of controversy, only the *left* label is propagated.

In order to better describe the entire process, the PN graph shown in Fig. 5a will be used to illustrate the partitioning procedure. Figure 5b shows how the activities are allocated to the peers through the TopDown procedure. Following the above partitioning algorithm, activities 1, 2 (*main sequence, begin and receive*) and 24, 25 (*reply and main sequence end*) are assigned to the Peer Collector. Then, starting from the invoke activities marked as 7, 15 and 18 activities 7–11, with the E1 label, are assigned to Peer1, activities 15 and 16, with E2 label, are assigned to Peer2 and activities 18–23, with E3 label, are assigned to Peer3. Then, applying the BottomUp procedure, activities 3–6 are assigned to Peer1, 12–14 to Peer2, 17 to Peer3 as shown in Fig. 5c.

3.3 Sunflower Decentralized Execution

On the basis of these schemas, Sunflower enacts the federation of (*SWEA*) agents that has to be executed on the SBE nodes. The decentralized execution of the workflow is coordinate by tokens exchanged among the SBE platforms. Tokens contain the whole execution state, including all data gathered during execution. Each *SWEA* agent performs the portion of workflow assigned and determine which agent should be activated next.

SWEA agents adapt their structure moving over the Internet to position themselves in the nodes with low workload and where the Web services with the best performance are available. The framework provides support for the migration-transparent of the agents and instructs the agents, by a migration policy, to migrate in order to achieve goals like load balancing, performance optimization or guaranteeing QoS.

Sunflower monitors the QoS for Web services by Antares [15] and effectively self-adapts the workflow engines in response to changes in load patterns and server failures. Antares is able to disseminate and reorganize service descriptors by an ant-clustering algorithm and, as a consequence of this, it facilitates and speeds up discovery operations. Based on dynamic service performance evaluation, the services with similar or same metric are gather into clusters by Antares. Scheduling managers make a scheduling decision based on user QoS requirements and information in Antares. All member services in a cluster provide similar or the same QoS after service clustering. Consequently, the task scheduling involves two steps: initial cluster selection from service clusters and further service selection from the selected cluster.

To support workflow adaptation the *SWEA* agents are assisted by routing/scheduling *RA* agents and monitoring/analizing *MA* agents that interact with the Antares information system. The MA agent collects details about the performance metrics and workload of the Web service and when it detects a change, owing to external

events, it inserts a new Web service descriptor with the new information in the Antares virtual space and notifies the change to the RA agent. When the RA agent receives a notification about a modification of the class of QoS, it sends a query to Antares to discovery and select a descriptor of an equivalent optimal service. Then, Antares returns a reference to an end point handler for the selected service. Before executing the sub-workflow, the *SWEA* agent contacts the *MA* agent to verify whether the class of QoS of the service to invoke is respected. In the affirmative case, the *SWEA* agent invokes the service and performs the workflow task, otherwise it uses its migration-policy to decide its destination consulting the RA agent. The activities of the *MA* and RA agents are performed continuously.

Sunflower uses the scheduling algorithm executed by RA agents, to make these choices, using information provided by Antares. For each Web service, the RA agent schedules the SWEA agents queued in the local SBE. The scheduled SWEA agent checks whether the QoS of the service relied on the node of the network is less than that required. If affirmative, a request is sent to the Antares registry service to search for an equivalent service to replace. If the service exists and is available, the reference to the service is returned to the RA agent that uses this information to migrate the SWEA agent on the node where the service is localized. Otherwise, if there are no services available an activation request for a new virtual machine is sent to the Cloud [16] provisioner.

Before, a new VM is started the provisioner checks whether there is one VM with the QOS requested already, else a new VM is started. The VM continues the execution of the workflow and before invoking the next Web service on the Cloud the RA agent checks, by querying Antares, whether an equivalent service is available on the Internet. If affirmative, the activation token with the status information is transferred to the SBE node on the network that has the next service to invoke.

4 Integration of RESTful SO in Sunflower

IoT technology emerges from the recent research and technology advancements in the fields of embedded systems and wireless sensor networks [17]. In these contexts, a plethora of electronic objects has been developed that fulfills even more complex requirements. These objects span from simple sensors to more and more flexible and programmable objects. In addition, all the objects should be able to interact with each other and with the *services* on the internet in order to fully accomplish the IoT philosophy. These considerations suggest these objects should be wrapped with a standard well-established interface that also addresses the complex and proactive behavior leading to the concept of *SO*.

REST services could be a way to deal with standardization issues in an easy and lightweight way. Also, REST technology strongly relies on *IP reachability* which is a fundamental concept of IoT technology in the world of IoT researchers.

Our middleware uses a REST interface to wrap SOs and also supplies an *event-driven* engine that properly captures the context modifications in the physical part.

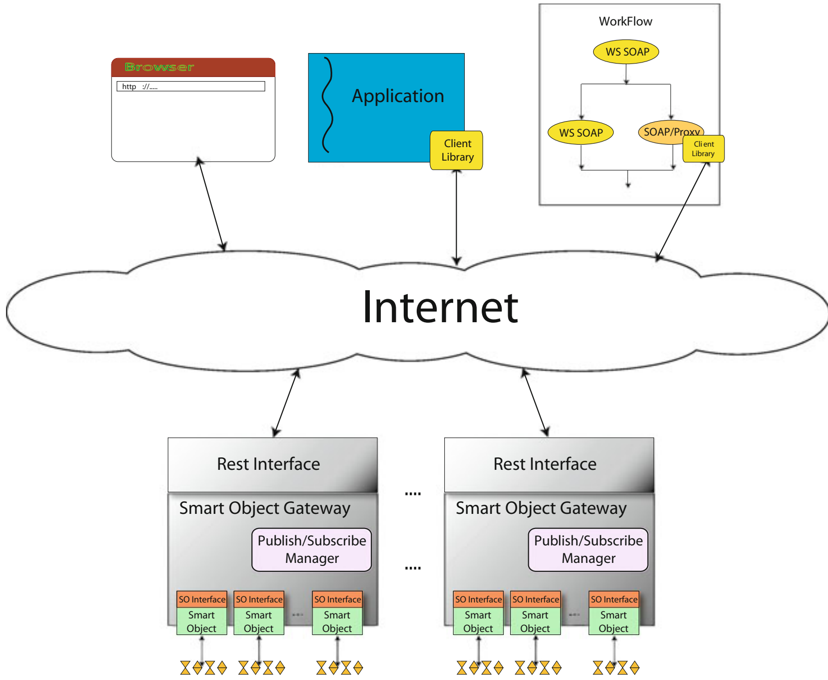

Fig. 6 Middleware architecture for integration of web services and smart objects

Roughly speaking, our middleware permits us to read/write the SOs and define events upon them through a well-established REST interface. Events are defined by *logical rules* submitted to the middleware and exploited in a *publish/subscribe* [18] fashion.

This approach is designed to ensure *ubiquity* and *location transparency* of SOs while fostering a *service-oriented* easy to use development of IoT applications.

As is shown in Fig. 6 our architecture was conceived so that SOs are linked to different computing nodes. Each SO is wrapped in a suitable `Smart Object Interface (SOI)`. All the SOs relative to a computing node, together with their Smart Object Interfaces, are managed by the SOG.

SOG represents the "glue" between the REST part and the SO part of the system. More in particular, SOG is a *singleton* (i.e. one SOG per computing node) and persistent entity of the middleware. Through SOG, REST invocations, which are intrinsically non-persistent and stateless, can access properly to the SOs which are conversely persistent and stateful. In addition, SOG is in charge of managing events defined by rules involving more than one SO; In this case, SOG divides an entire rule into different sub-rules and then assigns each sub-rule to the suitable SO.

As can be seen in Fig. 6 remote accessing of the SOs can be done in different ways: owing to REST protocol, one can access the middleware and relative SOs by using a normal browser, that is, by means of *HTTP protocol*. In a different scenario, one can interact with the middleware using an *ad hoc* client library that hides REST

invocation details offering suitable API for developing the application in general purpose programming languages such as: java, c++ and so on.

The third scenario is the one which is more of interest in the context of this work: it foresees full integration between the SO paradigm and web service's orchestration with the addition of an event-driven methodology. Web services orchestration could be realized in different ways, in the context of this work we propose the use of *WS-BPEL/SunFlower* described in the previous sections.

WS-BPEL technology permits web services to be orchestrated through workflow design. Nowadays, many graphical tools exist that allow WS-BPEL workflows to be developed in an easy manner in order to allow people, even with no skills in programming, to create their own applications. WS-BPEL relies on *SOAP* web services rather than on RESTs. Our middleware tackles this issue using *SOAP/REST proxy* services that permit full compatibility and integration.

4.1 SOs Versus Physical Resources

A SO can be composed of just a simple sensor or it can be a more complex object that includes many sensors, many actuators, computational resources like CPU or memory and so on. Examples of complex SOs can be: *smart room*, *smart flat*, *smart building* etc.

In general, SO outputs can be represented by *punctual values* (e.g. the temperature at a given point of a room) or *aggregate values* (e.g. the average of moisture during the last 8 h). Also, the values returned by SOs could be just the measurement of sensors or could be the result of complex computations (e.g. the temperature of a given point of space computed by means of interpolation of the values given by sensors spread across the environment). Furthermore, a SO could supply actuation functionality by changing the environment on the basis of external triggers or internal calculus.

These different kinds of behavior that SO can expose must be reflected in how it is integrated in the middleware. SO is therefore conceived as a complex object that can read and write upon many simple physical resources. More in details, we consider that each SO exposes different *functionalities*. Each functionality can be either sensing or actuating and can be refined by further parameters that dynamically configure it. The previous assumption leads to the definition of *simple physical resource* as the following *triplet*:

$$[SOId, SOFunctionId, Params]$$

where `SOId` uniquely identifies the SO, `SOFunctionId` identifies the specific functionality and `Params` is an ordered set of parameter values that configure the functionality. For example let's consider a *Smart Room* made of sensors for measuring different physical quantities inside a room such as *temperature*, *moisture*, *brightness* and so on. Suppose now you want to read from Smart Room the temperature in a given spatial point of the room. In that case the triplet could be:

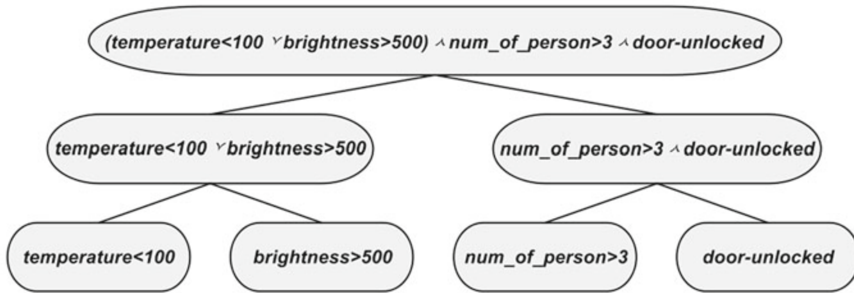

Fig. 7 Example of binary tree of a rule

$$[SmartRoom, temperature, [x, y, z]]$$

where x, y and z are the cartesian coordinate of the point of interest.

4.2 Publish/Subscribe of Events

SOG include a *publish/subscribe* component for managing events in each computing node. Each event is defined by a *logical rule* where one or more SOs could be involved. Each rule is a *logical proposition* in which the *atomic predicates* can be of the following kinds:

simple_physical_resource < *threshold* (*e.g. temperature* <*300*)
simple_physical_resource > *threshold*
boolean_resource (e.g. the_door_was_unlocked)

Just an example of rule:

(temperature < *100 and brightness* > *500*) *or number_of_ person* > 3 *or door_unlocked*

The SOG (specifically its publish/subscribe manager) is in charge to parse the logical rule and generate a *binary tree* made as explained below: each node N of the tree corresponds to a logical proposition $N()$. given L and R, the child nodes of N, their associated logical propositions are respectively $L()$ and $R()$ so that it results either *N() = L() or R()* or *N() = L() and R()*. The radix of the tree corresponds to the entire rule while the leaves contain the atomic propositions that SOG considers in order to pass them forward towards the suitable SOs.

An example of a binary tree representation of a composed rule is shown in Fig. 7.

A SO is in charge to establish each time when the assigned atomic propositions are *true* or *false*. The logical proposition of a given node is computed on the basis of the value of its child nodes. The root of the tree is recursively involved by this

bottom-up computation. As soon as the value of the root node (i.e. the value of the entire rule) changes SOG notifies all the subscriber.

4.3 Smart Object Interface

The previous sections are focused on supplying a sort of general formalization to SO and their relative functionalities. The effort of formalization is now useful to introduce a well-established interface for SO:

```
interface SmartObjectInterface {
   SOResult checkValue(SOFunctionId functionId, SOFunctionParams
       params);
   SOResult acting(SOFunctionId functionId, SOFunctionParams
       params);
   void setRule(SOFunctionId functionId, SOFunctionParams params,
        Operator operator, SOValue threshold, RuleMatchedListener
        listener);
   void setRule(SOFunctionId functionId, SOFunctionParams params,
        RuleMatchedListener listener);
}
```

Where the `checkValue` is the method for read a particular physical resource, `acting` is the method for performing an operation that produces a change in the smart environment. There are 2 methods `setRule`: the first concerns publishing of threshold based rules while the second is thought for Boolean resources (see Sect. 4.2).

The parameters of the methods follow the previously described logic: `functionId` identifies a functionality that the SO exposes, `params` is an ordered set of parameter values that configure the functionality, `operator` is just the comparative operator to be used for the rule, its value can be either $<$ or $>$, `threshold` is a numeric value intended as the threshold value of the rule. The last parameter of both `setRule` methods is a *listener* object that the SO have to notify when the value of a rule is changed. The involved SO will execute notifications by calling the methods of the `RuleMacthedListener` presented below:

```
interface RuleMatchedListener {
   void ruleMatched();
   void ruleNotMatched();
}
```

For instance, let's consider the previously introduced Smart Room example, if one wants to publish an event that occurs when the *temperature* in the [4,4,5] point of the room's space is less than 27 then the method `setRule` should be called as shown in the following code excerpt:

```
//pseudo code
SmartRoom.setRule(temperature_Id, [4,4,5],Operator.lessThan, 27,
    objectListener);
```

It is worth noting that the `SO Interface` is used only by the SOG while it is completely hidden at application level. SOG is in charge of interacting with the suitable SO on the basis of the application part that, in turn, interacts with SOG through the SOG interface presented in the next section. Finally, all the SOs will have to link itself to the suitable SOG calling the `register` method on the SOG thus supplying its unique *Id*. For example:

```
//pseudo code
SmartObjectGateway.register(SmartObjectId, this);
```

4.4 Gateway Interface

The SOG implements the `GatewayInterface` described below. This interface is exposed outside by means of REST technology. The middleware foresees a suitable *proxy* SOAP web service that executes REST invocations in order to reproduce the `GatewayInterface` in the client side thus permitting fully integration with WS-BPEL/SunFlower workflows.

```
interface GatewayInterface {
   void resourceNaming(String name, SOId soId, SOFunctionId
       functionId, SOFunctionParams params);
   SOResult check(String name);
   SOResult check(String name, SOFunctionParams params);
   SOResult acting(String name);
   SOResult acting(String name, SOFunctionParams params);
   void setRule(Rule rule, String idRule);
   void subscribe(String idRule);
}
```

The method `resourceNaming` assigns an identification `name` to a given resource supplied by a given SO. A resource is a particular instance of a *functionality* of a SO refined by some *parameters*. In other word, a resource is the above-mentioned triplet: [*SOId*, *SOFunctionId*, *Params*]. The `name` assigned to a resource via `resourceNaming` can be used in the other methods in order to simply identify the resource. Furthermore, the identification `name` of a resource is useful to compose the rules in a more human-readable fashion.

The method `check` reads the current value of the resource identified by `name`. `acting` triggers tha actuation operation upon the resource identified by `name`. Both `check` and `acting` methods are of two kinds: the first take only `name` as parameter and refers to the resource as it is previously defined in `resourceNaming`; the second kind, instead, permits to dynamically refine the parameters of the referred resource. The method `setRule` permits a complex rule to be published (e.g. (*tem-*

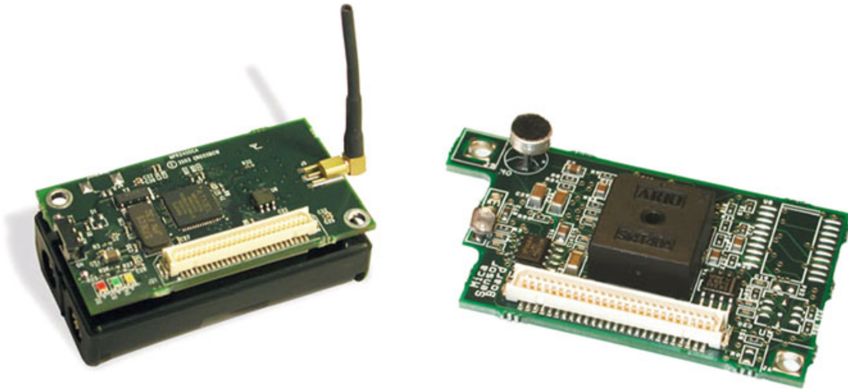

Fig. 8 Micaz mote and MTS310 sensor board

perature < 100 and brightness >500) *or number_of_person > 3 or door_unlocked*) and to assigns an id (i.e. `idRule`) useful for subscribing the rule afterwards.

The method `subscribe` permits a previously published rule to be subscribed that is identified by `idRule`. It is worth underlining that the latter method is a blocking method: when it is invoked, the middleware takes care of the event of interest while the execution of `subscribe` is stopped for waiting for the event to occur. This blocking behavior guarantees a correct integration in WS-BPEL/SunFlower workflows. In general, indeed, a workflow has one start point and one end point. Between start and end points there is the entire workflow that can be as complex as required. Nevertheless, it does not have any other entry points of execution except for the start point, so workflow cannot manage asynchronous operations properly because some sort of request/callback mechanism is required to cope with the asynchronous scenario.

5 Example of Usage

In this section we introduce a simple example of usage of the middleware in order to explain in detail how the proposed middleware works. In the example we use *Micaz motes* (Crossbow MPR2400) (see Fig. 8) as reference technology to build SOs.

A Micaz mote is a processor/radio board that run the operating system *TinyOS* [19], which handles power, radio transmission and networking transparent to the user. The Micaz system is widely used in the context of wireless sensors network where multiple motes distributed over a wide area are able to wirelessly transmit their data back to a *base station* attached to a computer.

TinyOS operating system enables Micaz motes to be programmed in *NescC* language supplying the chance to perform even complex elaborations directly inside the mote itself. Each mote can be expanded by attaching it a *sensor board* like MTS310

(see Fig. 8) which includes different kinds of sensing operations concerning physical entities such as *temperature*, *brightness* etc. In addition, each MTS310 sensor board includes 3 *led* that will be used as actuators in the context of our example. Firstly, we need to define the SOs we want, the functionalities they would offer and the meaning of the latter. After, we have to program the Micaz motes properly and develop suitable computer side SOs implementations that interact with motes via base station. In our simple example we define a single SO called `smart micaz` made of 3 Micaz motes. The first 2 motes both have sensing behavior while the third have the role of actuator. More in particular, the foreseen functionality for the first 2 concerns brightness while the third one exposes an actuating functionality that corresponds just to turn on and off a led. Formally, we call `light 1` and `light 2` respectively the sensing functionality of the first 2 motes and we call `actuator` the acting functionality of the third one. On the basis of the just defined functionalities we introduce the *resources* that will be used by the application part. Each resource corresponds to a triplet as explained in the previous sections:

```
light sensor 1 = { smart micaz, light 1, [] }
light sensor 2 = { smart micaz, light 2, [] }
led off = { smart micaz, actuator, [led = 0] }
led on = { smart micaz, actuator, [led = 1] }
```

Now we are free to use the resources as we want, we can read and publish/subscribe events upon `light senor 1` and `light sensor 2` or trigger actuation of `led off` and `led on`. The application we want to develop as example has a straightforward behavior: when the brightness sensed by the first mote (i.e. `light sensor 1`) decreases under a given threshold value, the led upon the third mote shall be turned on (i.e. triggering `led on`) whilst if it is the brightness measured by the second mote (i.e. `light sensor 2`) which decreases under the threshold value, the led upon the third mote shall be turned off (i.e. triggering `led off`). The above-described application is created by setting up a WS-BPEL workflow as shown in Fig. 9.

As can be seen the *flow* construct is used which enables executing commands in parallel. In our example there are two branches (i.e. sub-workflows) executing concurrently, each execution branch is in charge of controlling one of the reading resources (`light sensor 1`, `light sensor 2`) and triggering one of the writing resources (`led on`, `led off`). More in particular, each branch contains a *while* construct that loop infinitely. For each iteration a first proxy web service is called in order to subscribe an event defined by a rule such as `light sensor 1` < `150000` (the `Assign` construct passes the rule as parameter to the `Invoke` construct). The `subscribe` operation waits for the event to occur. When the rule is matched the first proxy web server ends its execution and a second proxy web server is called in order to trigger the actuation part. In that case the `Assign` construct passes `led on` or `led off` as a parameter to the `Invoke` construct.

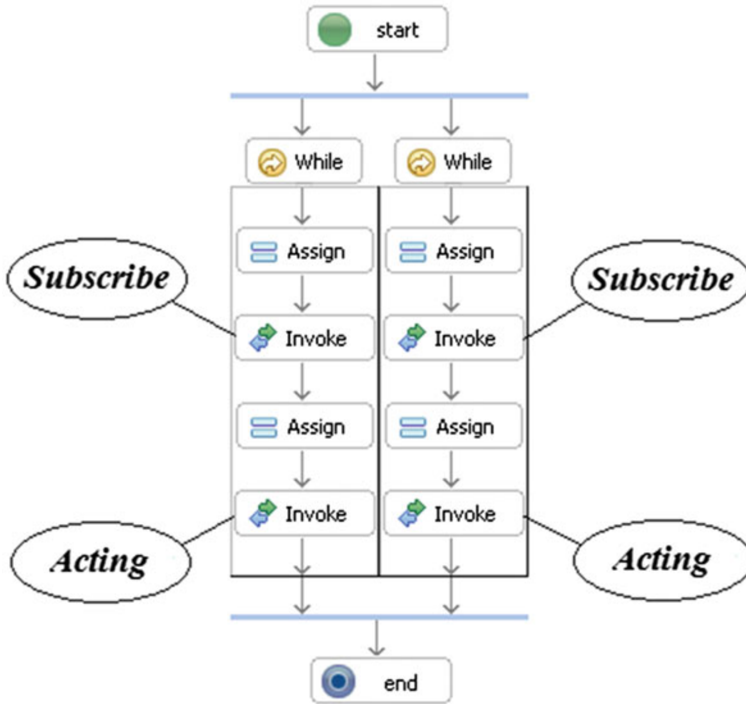

Fig. 9 WS-BPEL workflow

6 Conclusions

This chapter introduces an innovative framework and applications to combine Web service and SOs. The platform designed provides the execution of event-driven choreographic workflows. We present the design requirements and the corresponding architecture with a description of the technologies and platforms we intend to use for the implementation. Practical experience gained with the evaluation and implementation of the architecture demonstrates that it is both feasible and flexible to adapt to a variety of applications and off-the-shelf technologies. Regarding the implementation issues, we have shown that Event-driven Architecture has evolved to Event-driven SOA and this combination may form the foundation of emerging smart systems. Furthermore, we describe an implementation of a simple example of usage of the middleware following the SO's vision. In the near future, we aim to implement and test the proposed solution with the help of a smart room application.

Acknowledgments This work has been partially supported by TETRis—TETRA Innovative Open Source Services, funded by the Italian Government (PON 01-00451).

References

1. ISTAG Working Group Report on We-based Service Industry, February 2008, ftp://ftp.cordis.europa.eu/pub/ist/docs/web-based-service-industry-istag_en.pdf.
2. Alvarez, F., et al. (eds.): The Future Internet—Future Internet Assembly 2012: From Promises to Reality. LNCS, vol. 7281. Springer, Berlin (2012)
3. Atzori, L., Iera, A., Morabito, G.: The internet of things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010). ISSN 1389–1286
4. Kortuem, G., Kawsar, F., Fitton, D., Sundramoorth, V.: Smart objects as building blocks for the Internet of things. IEEE Internet Comput. **14**(1), 44–51 (2010)
5. Fortino, G., Guerrieri, A., Russo, W., Savaglio, C.: Middlewares for Smart Objects and Smart Environments: Overview and Comparison, in Internet of Things based on Smart Objects: technology, middleware and applications. Springer Series on the Internet of Things (2014)
6. Fortino, G., Guerrieri, A., Lacopo, M., Lucia, M., Russo W.: An agent-based middleware for cooperating smart objects. In: Highlights on Practical Applications of Agents and Multi-Agent Systems, Communications in Comp. and Inform. Science (CCIS), vol. 365, pp. 387–398. Springer, Berlin (2013)
7. Fortino, G., Guerrieri, A., Russo, W.: Agent-oriented smart objects development. In: Proceedings of IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 907–912 (2012)
8. Teixeira, T., Hachem, S., Issarny, V., Georgantas, N.: Service oriented middleware for the internet of things: a perspective. In: Proceeding ServiceWave'11 Proceedings of the 4th European Conference on Towards a Service-Based Internet, pp. 220–229. Springer, Berlin (2011)
9. Barker, A., Besana, P., Robertson, D., Weissman, J.B. : The benefits of service choreography for data-intensive computing. In: CLADE '09 Proceedings of the 7th international workshop on Challenges of Large Applications in Distributed Environments. ACM, New York, pp. 1–10 (2009)
10. Papuzzo, G., Spezzano, G.: Processing applications composed of web/grid services by distributed autonomic and self-organizing workflow engines. In: Chapman, B., Desprez, F., Joubert, G.R., Lichnewsky, A., Peters, F., Priol, T. (eds.) Parallel Computing: From Multicores and GPU's to Petascale. Advances in Parallel Computing. IOS Press, vol. 19, pp. 195–204 (2010)
11. Rahman, M., Buyya, R.: An autonomic workflow management system for global grids. In: Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), IEEE CS Press, pp. 578–583 (2008)
12. Alves A., et al.: Web Services Business Process Execution Language (WS-BPEL) 2.0, OASIS, August 2006, http://www.oasis-open.org/committees/wsbpel, accessed 12 Feb 2010
13. Brazier, F.M.T., Kephart, J.O., Van Dyke Parunak, H., Huhns, M.N.: Agents and service-oriented computing for autonomic computing: a research agenda. IEEE Internet Comput. **13**(3), 82–87 (2009)
14. van der Aalst, W., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge (2002)
15. Forestiero, A., Mastroianni, C., Spezzano, G.: Antares: an ant-inspired P2P information system for a self-structured grid. In: BIONETICS 2007–2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems, Budapest, Hungary, 2007
16. Buyya R., Yeo C.S., Venugopal S.: Market-oriented cloud computing: vision, hype, and reality for delivering IT services as computing utilities. In: Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008), IEEE CS Press, Los Alamitos, CA, USA, Sept. 25–27, 2008, Dalian, China. - Keynote Paper, 2008
17. Raghavendra, C.S., Sivalingam, K.M., Znati, T. (eds.): Wireless Sensor Networks. Springer, Berlin (2004)
18. Eugster, P., Felber, P.A., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. J. ACM Comput. Surv. (CSUR) **35**(2), 114–131 (2003)
19. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K.: TinyOS: An Operating System for Sensor Networks. Springer, Berlin (2005)