Information Fusion 21 (2015) 100–113

Contents lists available at ScienceDirect

Information Fusion

journal homepage: www.elsevier.com/locate/inffus

Architecture for management and fusion of context information

José M. Fernández-de-Alba *, Rubén Fuentes-Fernández, Juan Pavón

Dept. Software Engineering and Artificial Intelligence, Universidad Complutense de Madrid, Madrid, Spain

ARTICLE INFO

Article history:
Available online 13 November 2013

Keywords:
Context-awareness
Context-aware framework
Distributed blackboard model
Context aggregation
Context propagation

ABSTRACT

Information in a context-aware system has diverse natures. Raw data coming from sensors are aggregated and filtered to create more abstract information, which can be processed by context-aware application components to decide what actions should be performed. This process involves several activities: finding the available sources of information and their types, gathering the data from these sources, facilitating the fusion (aggregation and interpretation) of the different pieces of data, and updating the representation of the context to be used by applications. The reverse path also appears in context-aware systems, from changes in the context representation to trigger actions in certain actuators. FAERIE (Framework for AmI: Extensible Resources for Intelligent Environments) is a framework that facilitates management and fusion of context information at different levels. It is implemented as a distributed blackboard model. Each node of the system has a private blackboard to manage pieces of information that can be accessed by observer components, either locally or remotely (from other nodes) in a transparent way. The use of the framework is illustrated with a case study of an application for guiding people to meetings in a university building.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Last years have seen the proliferation of technologies for recognizing location, identity, movement, orientation, face and speech in a variety of devices. Especially relevant has been their integration in mobile devices, such as notebooks, tablets, and smart phones, and within facilities, for instance surveillance cameras and presence sensors in buildings. The combination of these resources gives rise to environments with numerous data information sources that can be dynamically assembled to solve high-level tasks.

Ambient Intelligence (AmI) [1] aims to integrate these technologies and devices to build *smart environments*. These environments combine information from sensors to infer the activities that are taking place at each moment, use that knowledge to anticipate users' needs, and build appropriate responses. In addition, smart environments are flexible enough to integrate new devices of different types and tolerate their potential failures without a noticeable configuration effort by users. This ability of systems to adapt themselves to the acquired *context* (both their own and that of users) is known as *context-awareness*. The *context* refers to any information related to people, places or objects that is relevant for the operation of applications [2]. It includes, for instance, preferences, current tasks, location and time, or available resources.

According to previous definition, it is essential for a *context-aware system* to facilitate integration of different kinds of information fusion processes. Information fusion is the merging of information from heterogeneous sources with different representations in order to obtain a more convenient or synthesized version of the information [3]. To develop this integration there exist numerous alternatives [4]. Among them, some are focused on the sources being combined [5], and other are focused on the data [6]. Each alternative has a certain impact on the adopting architecture.

Most architectural approaches to build context-aware systems use multilayered architectures [4,7,8]. These architectures facilitate the conceptualization and modularization of abstraction levels, but constrain the ability of components to work across multiple layers. They usually require complex management mechanisms for components, information and processes. Looking to overcome these limitations, researchers are considering alternative architectures with mechanisms that bring more flexibility and robustness. This is the case of works based on blackboard models [5]. However, this kind of models also presents some undesirable restrictions, for instance, the centralization in the management of context information.

FAERIE (*Framework for AmI: Extensible Resources for Intelligent Environments*) tries to overcome this issue by providing a distributed solution that implements a federated blackboard model. This model provides the view of a unique virtual blackboard for all components, hiding the details of the actual distribution of the system. This structure addresses several issues in earlier applications

* Corresponding author.
E-mail addresses: jmfernandezdealba@fdi.ucm.es (J.M. Fernández-de-Alba), ruben@fdi.ucm.es (R. Fuentes-Fernández), jpavon@fdi.ucm.es (J. Pavón).

1566-2535/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.inffus.2013.10.007

Table 1
Comparison table.

Systems/Frameworks	Context acquisition	Context modeling	Context processing	Distribution/Layers
iRoom [5]	Blackboard	Key-value tuples	Data-centric	Central server/dynamic
Context toolkit [6]	Widgets	Key-value tuples	Process-centric	p2p/Static
Context Management Framework [17]	Blackboard	Ontology-based	Data-centric	Central server/dynamic
Hydrogen [18]	Context server/point-to-point	Object-oriented	Process-centric	Central server/static
Haya's [19]	Blackboard/context graph	Key-value tuples	Data-centric	Central server/dynamic
OCP [20]	Virtual blackboard	Ontology-based	Rule-based	p2p/Static
Henricksen's [21]	Context server	Ontology-based	Process-centric	p2p/Static
Ejigu's [22]	Context server	Ontology-based	Process-centric	p2p/Static
Hermes [23]	Context server	Object-oriented	Process-centric	p2p/Dynamic
Venturini's [24]	Context broker	Ontology-based	Process-centric	p2p/Dynamic
FAERIE	Blackboard	Object-oriented	Data-centric	p2p/Dynamic

of blackboards to manage context, such as the dependency on a central server and the considerable network traffic generated. Besides, it provides to context-aware components a flexible access to information at any level of abstraction. It uses a subscription mechanism that is independent of the logical organization of systems in layers. With it, components can interpret and combine any intended information, either to increase or decrease its abstraction level. This mechanism facilitates the implementation within the components of the mentioned information fusion procedures. Moreover, since this architecture provides transparency for most of the issues related to distributed context management, it allows engineers to focus on the development of business logic, and reduces the errors not related directly with it. This will be illustrated with the case study of an application for guiding people in a building for a meeting.

The rest of the paper is organized as follows. Next two sections describe the FAERIE framework, with Section 2 presenting the architecture and Section 3 illustrating its use through the implementation of the case study. Section 4 provides a review of other context-aware architectures and compares them with our work. Finally, Section 5 draws some conclusions and issues on the evolution of the FAERIE framework.

2. The FAERIE framework

FAERIE conceives a *context-aware system* S as a set of interconnected *environments* $\varepsilon_1, \ldots, \varepsilon_n$. Each environment ε_i comprehends:

- A *physical space* p_i, which contains a single *main device* d_{i0} and a set of *peripheral devices* $d_{i1}, \ldots, d_{im}$ physically connected to d_{i0}. These devices include the sensors and actuators deployed in the space, as well as other elements that may provide additional services (e.g. a database connection). The type P is defined as the set of all possible combinations of states of devices in a physical space. The *location and coverage* of p_i are determined by the location, type, and range of the devices with sensing or acting capabilities in it, and it may change over time. This way, many physical spaces from different environments may overlap.
- A *computational space* c_i (also called *node* in a computer network), which is represented by the software framework running on d_{i0}. It contains the components implementing the logics of AmI applications. Concretely, it contains one *context container* κ_i holding the *context model*, and a set of *context observers* $o_{i1}, \ldots, o_{ip}$ that manipulate that model. The type K is defined as the set of all possible combinations of context elements in a context container.
- A set of *known environments* $\varepsilon_{i1}, \ldots, \varepsilon_{iq}$ such that each node $c_{i1}, \ldots, c_{iq}$ share a network with c_i.

Fig. 1 represents this structure. An example of connection among nodes is modeled in Fig. 2. It shows three main devices: a

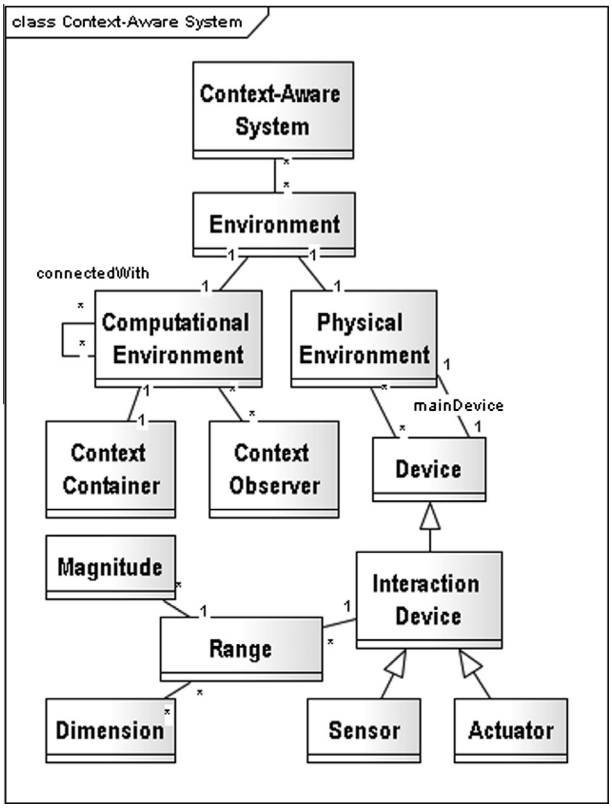

Fig. 1. Structure of a FAERIE-based context-aware system.

smartphone, a desktop computer, and a laptop. There are several peripheral devices (e.g., sensors and actuators), applications and files. Each main device runs its node, which contains its respective context container and context observers. Some of the context observers drive the peripheral devices.

A context container acts as a blackboard, and the context observers examine or update its information according to certain logic. When there is a change in the representation of the context, a notification is sent to the interested (i.e., subscribed) context observers, which are able to modify their behavior accordingly. This, in turn, may modify the current context or the behavior of the driven devices. Thus, the context observers are responsible of making the state of the context progress over time.

In a formal way, a general context observer o_j acts as a function of the form $o_j : K \times P \to K \times P$. This means, given a state of the physical environment and the context container, the observer generates a new state of the context container and changes the physical environment. Assuming a discrete division of the time,

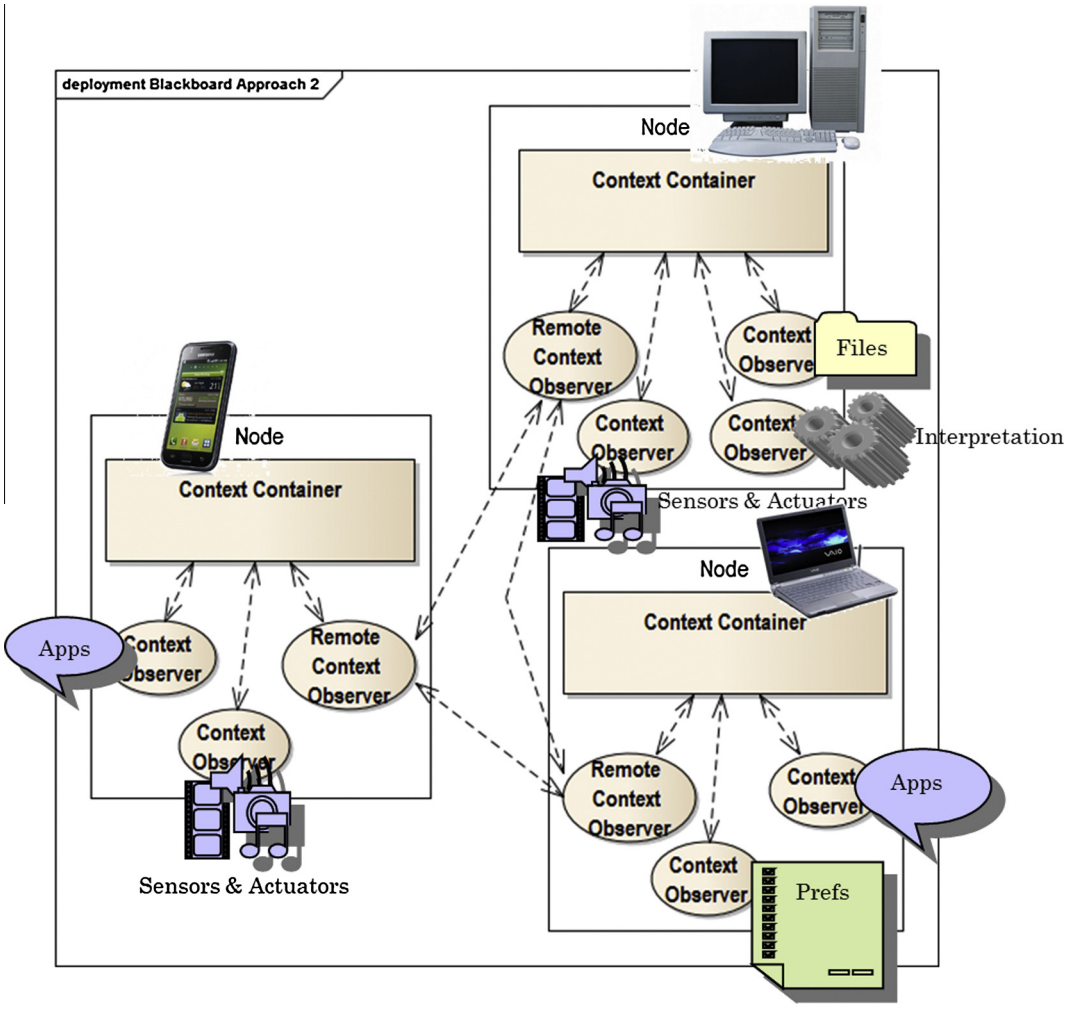

Fig. 2. Example of node connections as a distributed blackboard model.

Fig. 3 represents this process: $p(t)$ and $k(t)$ are the physical environment and the context state at time t; $o1,\ldots,o5$ are the context observers and $c1,\ldots,c6$ are the context elements. The state of the context container at a given time is obtained as a combination of the processes of the observers in the previous quantum of time.

Information fusion processes are one of the possible functions that context observers may implement. Concretely, their objective is to combine multiple sources of information in order to: provide a more synthesized and accurate version of the same information (aggregation), infer new information making use of heuristics or previous knowledge (interpretation), or disambiguate different versions of the same information from multiple sources [3].

FAERIE supports the development of systems that work in this way using mechanisms that are described in next subsections. The components are organized following a distributed data-centric blackboard model. This architecture is defined through the interfaces of *context container* and *context observer* components in Section 2.1. The context containers hold information that governs the behavior and interactions among the other components. Section 2.2 describes its structure. Context observers are able to register themselves with the context container of the node where they run, and to make requests on specific pieces of context. If a request cannot be solved locally within a node, it may be fetched on remote known nodes. Section 2.3 explains how this association between consumers and providers takes place at runtime. Finally, Sections 2.4 and 2.5 describe the framework support to implement the actual context processing. The former section is focused on sensing and acting mechanisms (i.e., direct interaction with sensors and actuators respectively), and the later on those to aggregate and propagate context information. This last section also explains the different types of process for context-information fusion that the framework facilitates.

2.1. Architecture

The software components running in a FAERIE *node* have to implement certain interfaces in order to collaborate. Fig. 4 shows their dependencies.

The *context container* component provides methods to access to the object-oriented representation of the current context (see Section 2.2). It also associates each element of that representation with the *context observer* components that work on them (see Section 2.3). The *context container* offers these services through the *ContextContainer* interface, and requires the *ContextObserver* interface to inform other components of context changes.

The *context observer* components request and react to changes in the representation of the current context. They can be classified as *grounded* and *abstraction context observers*. *Grounded context observers* access to the external environment using *peripheral devices*, either to modify it according to the context or to update the context according to it. More concretely, sensors can be seen as functions of the form $o_{sensor} : K \times P \rightarrow K$, and actuators as

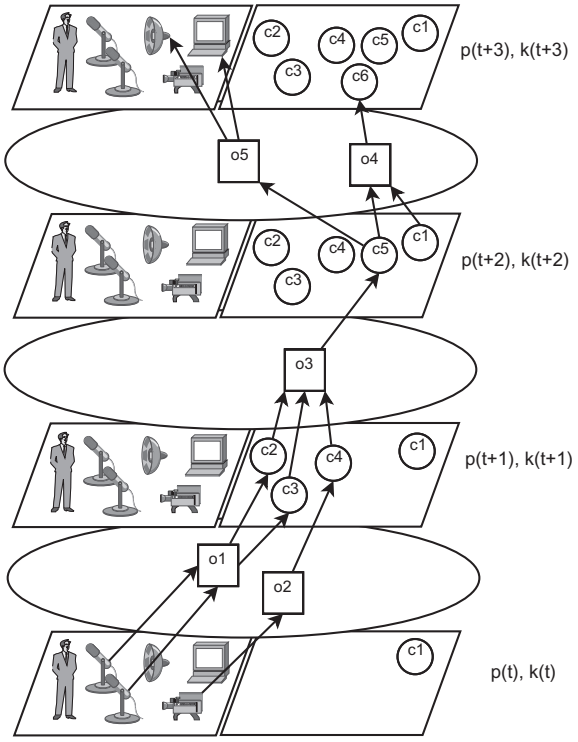

Fig. 3. Process of change of the context over time.

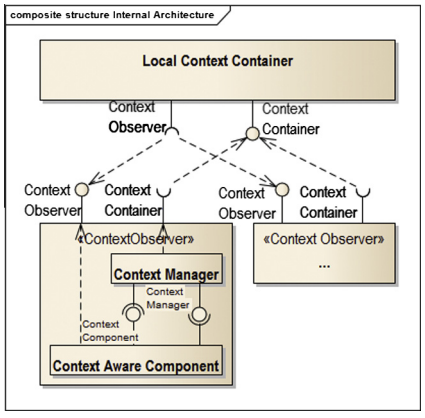

Fig. 4. Architecture of the framework components.

$o_{actuator} : K \rightarrow K \times P$ (see Section 2.4). *Abstraction context observers* do not access the environment, as they only update the context representation following their internal logic and using other information present in the context. In other words, abstraction context observers are of the form $o_{abs} : K \rightarrow K$ (see Section 2.5). Both types offer the *ContextObserver* interface and use the *ContextContainer* interface, which link them to *context containers*. They are internally divided into two subcomponents: the *context manager* and the *context-aware component*.

The *context manager* subcomponent implements the *ContextManager* interface to fulfill several responsibilities. First, it adapts the *ContextContainer* interface to offer the same functionality but in a simpler way. This relieves the *context-aware component* from some management issues, such as identifying itself in each information request, so it works as if it was the only component accessing to the context information. Second, it obtains and keeps track of the reference to the *context container* of its *node*. Since the *context container* is a dynamic component, it may be undeployed or changed during runtime. The *context manager* hides this to the

```
public <T extends ContextElement> T request(T element);

public boolean release(URI elementId);

public boolean subscribe(ContextObserver observer,
    ContextElement contextElement);

public boolean unsubscribe(ContextObserver observer,
    ContextElement contextElement);
```

Listing 1. Context access methods of the *ContextManager* interface.

context-aware component. Third, it contains the necessary code to launch the *context observer* in the *node*. This process provides the reference of the *context observer* component to the *context container*.

Listing 1 shows the methods of the *ContextManager* interface that allow the *context-aware component* to manipulate context information. The *request* method retrieves or publishes a certain *context element* in the *context container*. Given an instance of *ContextElement* as the method parameter, if the *context container* contains another instance with the same properties (e.g., class and id), that one is returned; otherwise, the provided instance is published on the *context container* and returned. Publishing means notifying each *context observer* able to *handle* the instance about this addition, in order to allow them to link to it as consumers or providers. Components use the *release* method to declare that they do not need an information element any more. If there is no other component interested in the same context element, the container unpublishes it. Unpublishing means notifying each *context observer* bound to the *ContextElement* about this removal. In this way, these observers can release resources that are no longer needed, for instance, stop monitoring a sensor. Context observers use the *subscribe* and *unsubscribe* methods to declare respectively that they are interested in updates of a certain *ContextElement* or that they are not any more.

The *context-aware* subcomponent implements two interfaces: *ContextObserver* to react to changes in the context; and *ContextComponent* to track its own lifecycle.

Listing 2 shows the methods of the *ContextObserver* interface. The *handles* method is invoked to determine if the component is able to handle the specified event. A *ContextEvent* may refer to the addition, removal or update of *ContextElement*s. This method can return 0 if it does not handle the event, or `HANDLE_R`, `HANDLE_W`, and `HANDLE_RW`. `HANDLE_R` means that the observer is not modifying the element, but it is interested in their changes; `HANDLE_W` implies that the observer modifies the element, but it does not read its state; `HANDLE_RW` is a combination of the previous two. When this method returns other than zero, the *elementAdded*, *elementRemoved*, and *elementUpdated* methods can be called depending on the result to notify the component of the corresponding events. This way, the *context-aware* components are able to track the changes in the context. These methods are used in different activities of the context-aware system, such as context discovering (see Section 2.3), sensing and acting (see Section 2.4) and context processing (see Section 2.5).

The *context manager* uses the methods of the *ContextComponent* callback interface (see Listing 3) to allow the *context-aware*

```
public int handles(ContextEvent event);

public void elementAdded(ContextEvent event);

public void elementRemoved(ContextEvent event);

public void elementUpdated(ContextEvent event);
```

Listing 2. Callback methods for context observing in the *ContextObserver* interface.

```
public void setContextManager(ContextManager contextManager);

public void activate();

public void deactivate();

public void unsetContextManager(ContextManager contextManager);
```

Listing 3. The *ContextComponent* interface contains the callback methods for lifecycle management.

component to react to changes in the lifecycle of the different components of the system. The *setContextManager* method is called when the *context container* becomes accessible, providing a reference to the *context manager* for the *context-aware component*. The *activate* method is called after the previous one to trigger the initialization code; the *deactivate* method is called before the shutdown of either the *context-aware component* or the *context container*, to trigger the disposal code. The *unsetContextManager* method is called right before the *context container* becomes inaccessible.

2.2. Context modeling

Among the alternatives to implement the representation of the context, FAERIE uses an object-oriented approach [9], based on the *observer* pattern. Fig. 5 shows the elements that can be found in the computational space of a FAERIE system. Those relative to the modeling of information are placed on the right half, while those relative to its processing are placed on the left.

At a given moment, each piece of context is represented as a *ContextElement* of one of three possible types: *Entity*, *Attribute* and *Relationship*. An *Entity* represents a concept of the system's domain vocabulary (e.g., a person, place or object), and groups all the concrete *Attributes* and *Relationships* related to it. An applied *Attribute* holds a piece of information of type *T* with certain meaning related to an *Entity* of type *E* (e.g., the age in years of a person, or the name string of a place). An applied *Relationship* links an *Entity* of

```
public synchronized boolean attemptUpdate(
    ContextValue<T> newValue) {
  boolean succeeded = false;
  // Notifies the attempt to the context
  if (this.getContext().publishValue(this, newValue)) {
    // If this attempt is accepted, the value is updated
    this.value = newValue;
    succeeded = true;
  }
  // Notify the observers
  this.notifyAllObservers(UPDATE_ATTEMPT, newValue);
  return succeeded;
}
```

Listing 4. Method for attributes.

type *S* to others of type *T* according to certain semantics (e.g., a person with the place where s/he is located). The current state of each *Attribute* and *Relationship* is stored as a *ContextValue*. A *ContextValue* contains, besides data of a certain type *T* (specified by a concrete *Attribute* or *Relationship*), a reference to the source *ContextObserver* (i.e., the context observer that produced the value), its creation timestamp (i.e., the moment when the value was obtained), and additional metadata (e.g., a number representing the quality or the cost of the measurement).

In a formal way, the contents of a *context container* κ at a given time t, $\kappa(t) \in K, \kappa(t) \subset ContextElement$ are of the form $\kappa(t) = E(t) \cup \mathcal{A}(t) \cup \mathcal{R}(t) \cup O(t)$, where:

- $E(t) \subset Entity$ is the finite set of referenced entities at time $t, e_1, \ldots c, e_r$.
- Let $A : Entity \times Attribute$ the set of pairs of entities and attributes. $\mathcal{A}(t) : A \to ContextValue$ is a function such that $\mathcal{A}(t)(e, \alpha) = v$ being v the value of the attribute α of the entity e at time t.
- Let $R : Entity \times Relationship$ the set of pairs of entities and relationships. $\mathcal{R}(t) : R \to [ContextValue]$ is a function such that $\mathcal{R}(t)(e, \rho) = [v]$ being $[v]$ the list of entities related with e by the relationship ρ at time t.

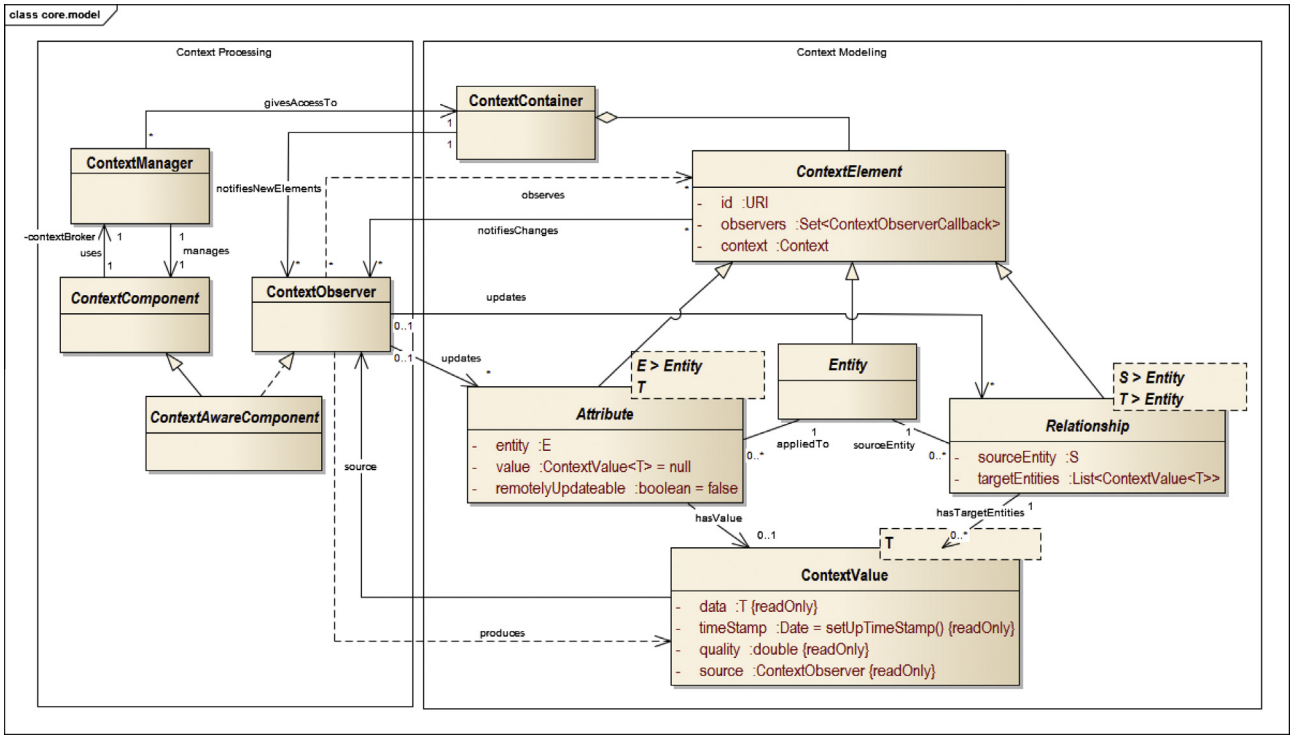

Fig. 5. Object-oriented model of the computational space.

- $O(t) \subset ContextObserver$ is the finite set of context observers existing at time t, $o_1, \ldots c, o_p$. Each observer o_j has a set of context elements that is *observing* at time t, $Read_j(t) \subseteq \kappa(t)$, and a set of context elements that is *updating* at time t, $Write_j(t) \subseteq \kappa(t)$.

The *ContextElements* provide a method to make an *update attempt* of their values. Listing 4 shows its pseudocode. Whether the attempt is successful or not depends on the consistency mechanisms implemented by the *context container*, as it will be explained in Section 2.5. This way, several context observers may have the same *Attributes* and *Relationships* in their $Write_j(t)$ set at a certain time, but only some of them will actually update their value. Both the attempts and the actual updates will generate events to the subscribed context observers. Using this information, each context observer is able to know what values are generating the others, and use them in their own processing.

As *ContextElements* are classic program objects, handling their information does not require using specific parsers or interpreters. This reduces the processing workload when compared, for instance, with ontology-based models, which are usually represented as XML documents. In addition, this basic model for context can be extended through inheritance, specifying new types of *Entities*, *Relationships* and *Attributes*. These new types may hold specific restrictions on the contained information and even specific methods with their own semantics.

2.3. Context discovery

Context discovery is the process that connects *context observers* that work on the same *context element*. This discovery can happen both when reading or updating information. In the first case, the consumer observer o_i requests some $c_a \in Read_i(t)$, so the system needs to find another observer o_j such that $c_a \in Write_j(t)$ (i.e., provides the required information). In the second case, the updater observer o_i requests some $c_a \in Write_i(t)$, so the system needs to find another observer o_j such that $c_a \in Read_j(t)$ (i.e., that propagates the changes, probably lowering their abstraction level). There are two levels of context discovery: *local*, when it happens within a node, and *remote*, which involves several nodes.

Fig. 6 shows an example of the process for *local context discovery*. A *context observer* that needs a context element makes a *request* to the *context container* by its *ContextContainer* interface. In this example, that request is for the attribute *TestAttribute* of a certain *TestEntity* (an example could be the *ValueDetected* of a certain *Sensor*). When the container receives the request, it has no previous element with the requested features. So, it includes the provided one into the context and tries to locate a suitable *ContextObserver* to provide its information. The container looks for this *ContextObserver* among those registered with it. It invokes the *handles* method (see Section 2.1) of the observers using the new element as argument for the event. The container selects the first observer

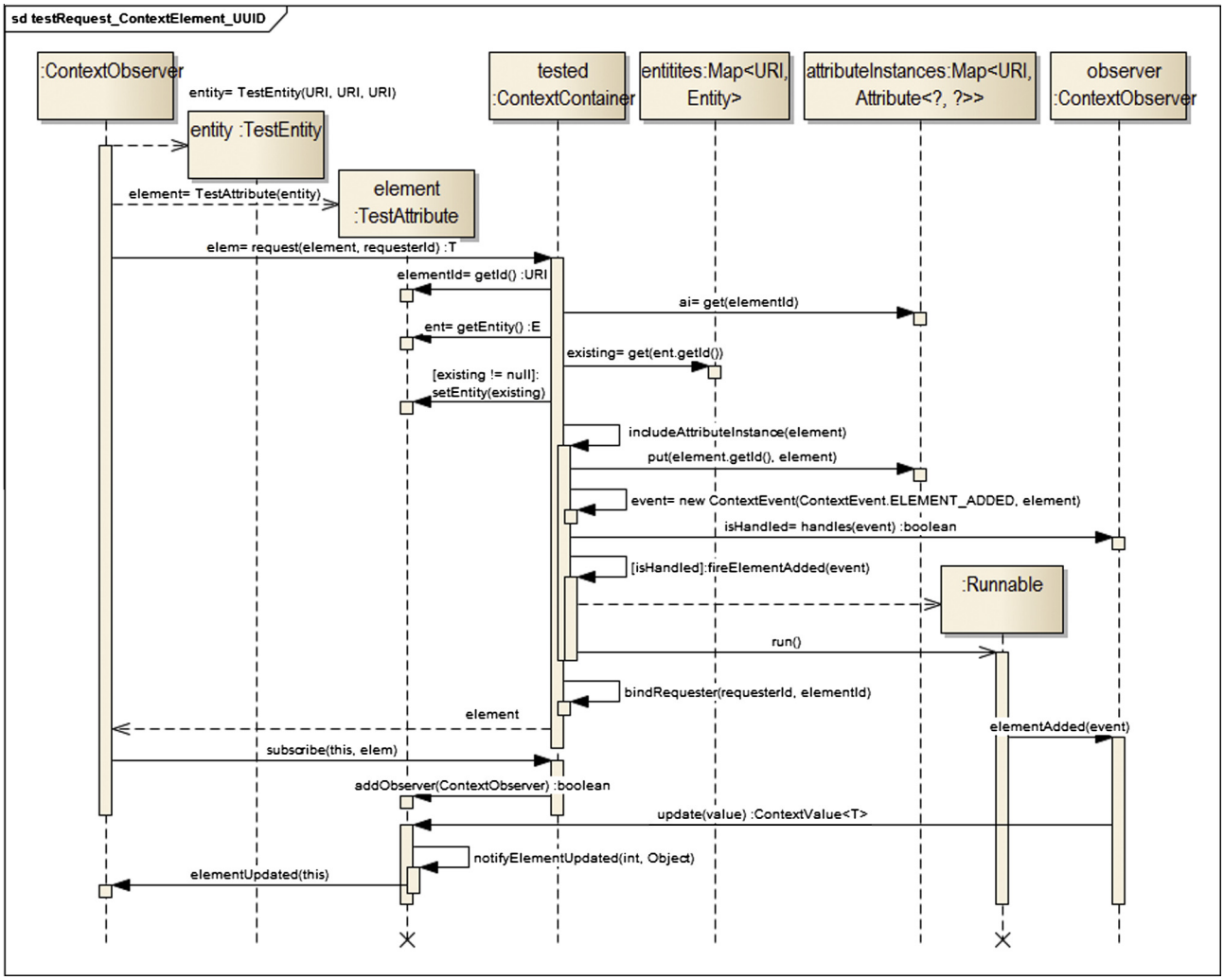

Fig. 6. Binding mechanism between two local *context observers*.

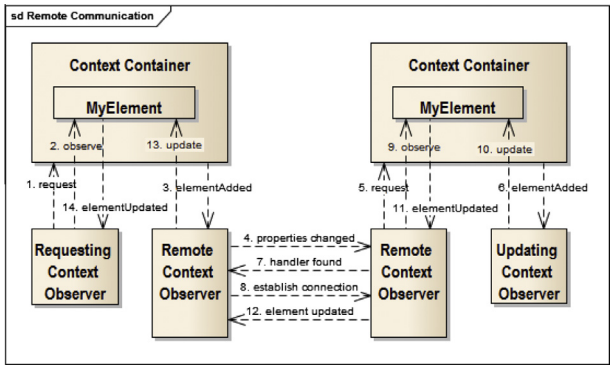

Fig. 7. Binding mechanism between two *remote context observers*.

```
public void elementAdded(ContextEvent event) {
  final Attribute attrib = event.getAttributeInstanceArg();
  thread = new Thread(new Runnable() {

    public void run() {
      while (true) {
        // Obtain attribute value from the environment
        // ...
        if (/* value has changed */)
          attrib.update(new DefaultContextValue(value, 1.0));
      }
    }
  });
  thread.start();
}

public void elementRemoved(ContextEvent event) {
  thread.interrupt();
}
```

Listing 5. Typical code for implementing a *context observer* driving a sensor.

whose method returns `HANDLE_W` and binds it to the element using its *elementAdded* method (see Section 2.4 for more on this). From that moment on, the bound observer can *attempt update* the *ContextElement* whenever it is necessary. The *ContextElement* will directly notify these updates to all the *ContextObserver*s subscribed to these events (see Section 2.4). In the case that the container is configured to do so, it will forward the request to the *remote context observer* of the *node* if needed. Also, the newly registered *ContextObserver*s will be notified about the existence of the context element.

A *remote context observer* performs two specific tasks upon its activation. It publishes to the network a reference to itself and detects the references and properties of other *remote context observers* in the network. These properties describe the *ContextElement*s that those *remote context observers* expect others to handle. This exchange of references and properties supports the *remote context discovery* that allows *remote context observers* to act as proxies for each other.

Fig. 7 depicts the process of *remote context discovery*. It starts when a request has been forwarded to a *remote context observer*. This observer modifies its registered properties to indicate the *ContextElement* that it needs to be handled. Then, it broadcasts these changes to the rest of *remote context observers*. These observers request that *ContextElement* to their own *context containers*. If some of these *nodes* contain a component able to handle that *ContextElement*, their *remote context observers* notify it back to the original requester. This chooses one (or many) of them and establishes a connection between the local and remote copies of the *ContextElement*. From that moment on, every change in the local or remote *ContextElement* will be forwarded to the other through the *remote context observers*.

The use of *remote context observers* makes transparent whether local or remote components update context. The *remote context observer* acts as any other observer in local interactions, while it effectively handles every context element that must be processed remotely by collaboration with other *remote context observers*. This connection creates a "virtual overlapping" among context containers, allowing the system to work as if it was connected to a unique virtual distributed blackboard.

The underlying protocol used to implement remote references and property broadcasting is Zeroconf [10] (a.k.a. *Rendezvous* or *Bonjour*), which uses DNS (Domain Name System) mechanisms. Zeroconf allows publishing and discovering remote objects with associated properties in the local network. It does not require any special configuration of the network parameters. This protocol is implemented transparently by R-OSGi [11] and web services, which have been used in the example implementations of FAERIE. Due to limitations of the Zeroconf protocol, the mechanisms described in this section only work when *nodes* are located within the same local network. When this is not the case, it is necessary to consider a different protocol.

2.4. Sensing and acting

Sensing and acting are both extremes of the external interaction of the context-aware system with the environment. Sensing refers to processing new data from sensors and acting to sending new commands to actuators. From the FAERIE perspective, both processes are very similar. They start as part of a context discovery process (see Section 2.3), when the container finds a suitable provider *grounded context observer* for a new *ContextElement*. Then, these processes just differ in which component updates the *ContextElement* and which observes it.

For *sensing*, the selected *context observer* drives a sensor. The container binds that observer to the *ContextElement* using its *addedElement* method. This call effectively starts the monitoring of the sensor (see Listing 5). The observer creates a thread where it checks changes in the sensor value, and updates the *ContextElement* related to it when these changes happen. In the same way, the *elementRemoved* callback is used to notify the observer that the *ContextElement* is no longer needed, and hence it stops its monitoring.

In a formal way, the process of sensing acts as a function of the form $o_{sense}(t) : K \times P \rightarrow K$ such that $o_{sense}(t)(\{(e, \alpha, \_)\}, p) = \{(e, \alpha, v)\}$ where v is the value provided by the context observer for the attribute α of the entity e under the physical environment state p at time t. The entity e represents a physical device d driven by the observer.

The process of *acting*, as that of sensing, starts with a new *ContextElement*, but here, the process triggers a command for an actuator. When a provider *context observer* publishes some of these

```
public void elementAdded(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  contextManager.subscribe(this, attrib);
}

public void elementUpdated(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  ContextValue value = attrib.getValue();
  // Check conditions on the obtained value
  // Perform the required actions
  // ...
}

public void elementRemoved(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  contextManager.unsubscribe(this, attrib);
}
```

Listing 6. Typical code for implementing a *context observer* driving an actuator.

ContextElements, the container search for a suitable *context observer* using the *handles* method. When found, it invokes the *elementAdded* method of the selected *context observer* (see Listing 6). As this last component drives an actuator, it does not update the *ContextElement*, but *observes* it instead. Then, when the *ContextElement* is updated, the actuator observer receives the *elementUpdated* callback (see Listing 6), and performs the action corresponding to the value obtained from the element.

In a formal way, the process of acting acts as a function of the form $o_{act}(t) : K \rightarrow K \times P$ such that $o_{act}(\{(e, \alpha, v)\}) = (\{(e, \alpha, v)\}, p)$, where p is the behavior executed by the context observer in the environment at time t as a response of the value v of the attribute α on the entity e.

2.5. Context processing

Context processing in FAERIE is defined as the transformation of the context model performed by one or more *abstraction context observers*. This transformation may flow in two possible directions: *bottom-up* and *top-down*.

A *bottom-up evaluation* starts when a *context observer* requests one or many *ContextElements* to *observe* their values, as in a *sensing* process (see Section 2.4). In this case, the provider *context observers* may need subsequent requests to the context, producing a cascade evaluation. The cascade evaluation finishes in *grounded context observers*. From these values, pending calculations are made backwards. These calculations can consist on *aggregation* or *interpretation* of values [4]. Fig. 8 shows an example of the runtime relationships that are established during this process.

Listing 7 shows an implementation of a context observer involved in a *bottom-up evaluation*. When a certain *Attribute* is published, the method *elementAdded* is called. The calculation of this *Attribute* needs the values of two more *Attributes*, *s1att* and *s2att* which in this case come from some sensors with identifiers *s1* and *s2*. These are requested and then observed by invoking the *request* and *subscribe* methods respectively. When any of these attributes change, the *elementUpdated* method is called. It aggregates their values to calculate the first *Attribute*. When this higher-level *Attribute* is removed from the context, the method *elementRemoved* is called. It releases the lower-level *Attributes* used before to calculate the other attribute.

In a formal way, the bottom-up evaluation done by an observer is a function of the form $o_{bottomup}(t) : K \rightarrow K$ such that $o_{bottomup}(t)(\{c_0, \ldots, c_n\}) = \{c_0, \ldots, c_n, c_{n+1}\}$ where c_{n+1} is the context element obtained by the context observer using the list of context elements $c_0, \ldots, c_n$ present in the context.

A bottom-up evaluation is called an information fusion process if takes one of the following concrete forms:

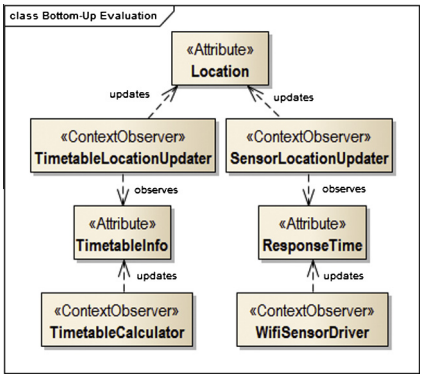

Fig. 8. Bottom-up evaluation.

```
public void elementAdded(ContextEvent event) {
  updatedAttribute = event.getAttributeInstanceArg();

  // Request the values of lower context elements
  s1 = new MySensor(/* ID of the sensor */);
  s1att = new MySensorAttribute(s1);
  s1att = contextManager.request(s1att);

  s2 = new MyOtherSensor(/* ID of the other sensor */);
  s2att = new MyOtherSensorAttribute(s2);
  s2att = contextManager.request(s2att);

  // ...

  // Observe the values of the lower sensors
  contextManager.subscribe(this, s1att);
  contextManager.subscribe(this, s2att);
  // ...
}

public void elementUpdated(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  if (attrib.equals(s1att))
    value1 = attrib.getValue();
  if (attrib.equals(s2att))
    value2 = attrib.getValue();
  // ...

  // Recalculate context value for the updated attribute,
     depending on the obtained values
  // ...
  updatedAttribute.update(new DefaultContextValue(value, 1.0));
}

public void elementRemoved(ContextEvent event) {
  contextManager.unsubscribe(this, s1att);
  contextManager.release(s1att);
  contextManager.unsubscribe(this, s2att);
  contextManager.release(s2att);
  // ...
}
```

Listing 7. Typical code to implement a bottom-up evaluation.

- **Aggregation:** this kind of processing takes multiple pieces of context information and produces a new context element that synthesizes all the information in a more condensed or abstract way. For example, building a global map using the adjacency information of different places belongs to this category. Typical aggregation techniques are machine learning techniques such as naive Bayes classifier (summarize large amounts of data as a set of classes defined by probability of certain features) [12].
- **Interpretation:** this kind of processing is similar to aggregation, but the result is new information that is inferred from the source information, rather than just a synthesized version of it. For example, guessing the location of some object using the values of certain sensors, and the information of their deployment. Subsumption inference is a typical interpretation technique (defining a concept as the conjunction of other more simple concepts) [13].
- **Redundancy/consistency handling:** this kind of processing involves handling different sources for the same information in order to produce a "better" value. The meaning of this "better" depends on certain policies of the system. For instance, if there are two components calculating the location of the same object using different algorithms and resources, this process may take one of the measurements, or a combination of both. Compression techniques are used to deal with this kind of problem [14].

The generated information is consumed by context-aware applications to perform *adaptation* (i.e., modify the external behavior accordingly). An example of this is guiding users in an area depending on their current locations.

The *top-down evaluation* works in the opposite direction. A *context observer* requests *changing* one or more *ContextElements*. That causes successive requests to *change* other *ContextElements*. These requests finish in *acting* processes (see Section 2.4). Then, each

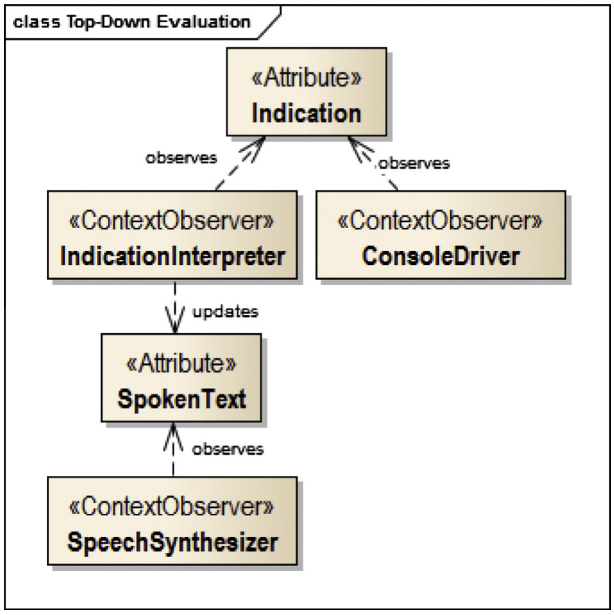

Fig. 9. Top-down evaluation.

```
public void elementAdded(ContextEvent event) {
  observedAttribute = event.getAttributeInstanceArg();

  // Observe the higher context element
  contextManager.subscribe(this, observedAttribute);

  // Request the values of lower context elements
  a1 = new MyActuator(/* ID of the actuator */);
  a1att = new MyActuatorAttribute(a1);
  a1att = contextManager.request(a1att);

  a2 = new MyOtherActuator(/* ID of the other actuator */);
  a2att = new MyOtherActuatorAttribute(a2);
  a2att = contextManager.request(a2att);

  // ...
}

public void elementUpdated(ContextEvent event) {
  observedValue = event.getAttributeInstanceArg().getValue();

  // Recalculate the actuators values depending on the new
     higher value
  // ...
  a1att.update(new DefaultContextValue(value1, 1.0));
  a2att.update(new DefaultContextValue(value2, 1.0));
  // ...
}

public void elementRemoved(ContextEvent event) {
  contextManager.release(a1att);
  contextManager.release(a2att);
  // ...
  contextManager.unsubscribe(this, observedAttribute);
}
```

Listing 8. Typical code to implement a top-down evaluation.

time the value of the original upper-level *ContextElement* is changed, these changes are notified downwards, ultimately activating the actuators. For this reason, the process is described as *top-down*. This defines a process of propagation of the values of *ContextElements* towards more concrete values. Fig. 9 shows an example of the runtime relationships that are set up during this process.

Listing 8 shows an example of implementation of a context observer for a *top-down evaluation*. When certain *Attribute* is published, the method *elementAdded* is called. It adds the *context observer* as an observer of the parameter higher-level *Attribute* using the *subscribe* method. This *Attribute* provides a value that has to be propagated to two other *Attributes* that depend on actuators. The *a1att* and *a2att Attributes* are requested, but the triggering *Attribute* is the observed parameter. When this original *Attribute* changes, the *elementUpdated* method is called, and its value is propagated to update the value of the other two *Attributes*. When the higher-level *Attribute* is removed from the context, the method *elementRemoved* is called. It releases the lower-level *Attributes* that depend on the value of the parameter *Attribute*.

In a formal way, the bottom-up evaluation done by an observer is a function of the form $o_{topdown}(t) : K \rightarrow K$ such that $o_{topdown}(t)(\{c_0\}) = \{c_0, c_1, \ldots, c_n\}$ where $c_1, \ldots, c_n$ are the context elements derived by the observer using the element c_0. As we can see, this process "spreads" the value v_1 of a_1 to a list of context elements, as it is the inverse process of bottom-up.

3. A development process for using the framework

This section illustrates the process to implement a context-aware application using FAERIE. For this purpose, it uses the case study of an application that supports a flexible tutorship program in a university. The goal of the application is a better use of time by lecturers and students. During tutorship hours, lecturers are allowed to work in tasks different than tutoring in any part of the building, as long as no student demands a meeting. On their part, students profit from longer tutorship timetables. This arrangement is only suitable if students can easily locate their lecturers when they need tutorship. When this happens, the system is responsible of inferring the location of both of them in the building, and guiding the student to meet the lecturer.

The process proposed to design this kind of context-aware application in FAERIE is divided into several stages: *considerations*, *identification*, *modeling*, *implementation*, *deployment* and *testing*. They are described in the next subsections. At the end, the results of the application of the architecture are discussed.

3.1. Considerations

The *considerations* stage is a preliminary phase in which the characteristics of the *environments* and the entities involved in the application are studied. The objective is to relate the requirements of the application to the actual conditions of the deployment scenario. These conditions include:

- *Features of physical environments*. For instance, battery use and computational capacity of mobile environments, size and distribution of spaces, and level of noise and characteristics of the network access.
- *Available components and capabilities*. Some examples to consider are physical components (e.g., peripherals, routers and sensors), software components (e.g., drivers and interpreters), and external services (e.g., web services for maps and components for external database access).
- *Users' characteristics*. The main factors here are the number of users and their preferences, including privacy issues and special needs if any.

In this case study, there are several potential elements of each category. This analysis considers only the most common ones.

Typical physical components of this scenario are the *mobile phones* of *students* and *lecturers*. Nowadays, it is quite usual that people in universities own a smart phone with a *network interface*. Such phones can be used to access the *application*, but also to provide *location* information to it. This information can be obtained through triangulation with *surrounding wireless antennas*. Therefore, these antennas are also physical components of this scenario.

Using wireless signals for triangulation is not a trivial task. Involved algorithms must deal, for instance, with signal noise, blind spots, and the addition and removal of antennas. These algorithms are already encapsulated and available in specific software components with the required learning and triangulation algorithms.

The previous location process needs that lecturers always have their phones with them. However, they can switch them off during a class or leave them in their offices. In this case, the context-aware system can resort to their public *timetables*: if the lecturer has a lecture at *current time*, s/he should be in the room where that activity takes place. This information is encapsulated for its use by another software component managed by the *university*.

3.2. Identification

The *identification* phase determines the key elements of the application based on the analysis of the *considerations* stage (see Section 3.1). The result is a list of *context elements* (see Section 3.2.1), and the rules that influence their changes (see Section 3.2.2).

3.2.1. Elements

The *considerations* stage allows identifying several elements that are expected to be relevant for the operation of the application. They constitute part of its *context* [2], and will be represented in FAERIE as *context elements* (see Section 2.2). Their list is subject to grow or shrink in further iterations of the process. In this phase, all the vocabulary of the system is just put together in order to take it into consideration. These elements are refined to their actual representations in the "Modeling" stage.

- *Application* (Entity)
- *University* (Entity)
- *Mobile phone* (Entity)
- *Student* (Entity)
- *Lecturer* (Entity)
- *Network interface* (Entity)
- *Wi-fi antenna* (Entity)
- *Surrounding Wi-fi antennas* (Relationship)
- *Response time* (Attribute)
- *Location* (Relationship)
- *Timetable* (Attribute)
- *Current time* (Attribute)

After further analysis of the identified elements, the list is refined with the following modifications:

- *University* is replaced with *Place* within university. The university as a whole is not suitable to be a "physical space" for the application. The spaces to consider are the locations that constitute the university campus.
- *Mobile phone*, *Network interface* and *Wi-fi antenna* are aggregated into a virtual sensor, called *Wi-fi sensor*. Each *Wi-fi sensor* provides the *Response time* of a single *Wi-fi antenna*, perceived by the *Network interface* of a certain *Mobile phone*. This simplifies the domain, as three entities are collapsed into one.
- *Surrounding Wi-fi antennas* is replaced with *Surrounding Wi-fi sensors* due to the previous modification.
- *Current time* is no longer considered as a *ContextElement*, as its calculation does not need any special treatment and it is globally accessible.

3.2.2. Rules

The design of the application also requires determining the dependencies between the *context elements* of the application. These dependencies guide the development, as the application needs components able to resolve them. At this step, the integration of fusion algorithms is essential, since most of the identified dependencies are resolved using information fusion techniques, as explained in Section 2.5. The identification of the following dependencies establishes the structure of the data to be combined. The dependencies for this case of study are:

- the *Application* **observes changes in**:
  - the *Lecturer*'s *Location* & the *Student*'s *Location*. This is an *adaptation* process.
- a *Person*'s *Location* **changes with**:
  - either the *Response time* of the *Surrounding Wi-fi sensors* for that *Person*. This is an *interpretation* fusion.
  - or *Current time* & that *Lecturer*'s *Timetable* (if that person is a lecturer). This is an *interpretation* fusion.

The framework will make a *redundancy handling* fusion of these two rules.

- a *Person*'s *Surrounding Wi-fi sensors* **change with**:
  - newly discovered *Wi-fi sensors*. This is an *aggregation* fusion.
- a *Wi-fi sensor*'s *Response time* is **sensed** (i.e., provided as seen in Section 2.4) by a given component.
- a *Lecturer*'s *Timetable* is **sensed** (i.e., provided) by a given component.

3.3. Modeling

The *modeling* stage represents the elements and rules identified in the *identification* stage (see Section 3.2) as a context model. This context model uses as modeling primitives *context elements* (see Section 2.2). If the *Entities*, *Relationships* and *Attributes* have been properly identified, the modeling just consists in representing them in a class diagram. Fig. 10 shows the result for the example application.

If there are other existing systems running in the *environments* of the application under study, it is advisable to try to adjust this model to fit the existing context models. In this way, it is possible to reuse already available *context observers* to handle some of the *ContextElements* of this application. For instance, Fig. 10 contains several superclasses that are reused from previous applications. These classes are *Person*, *SurroundingSensors*, *Sensor*, and *SensorValue*. This reuse reduces development effort and redundancy of components between applications. For this reason, the possible class *SurroundingWifiSensors* has been discarded. The *WifiSensors* will be just some of all the *SurroundingSensors*.

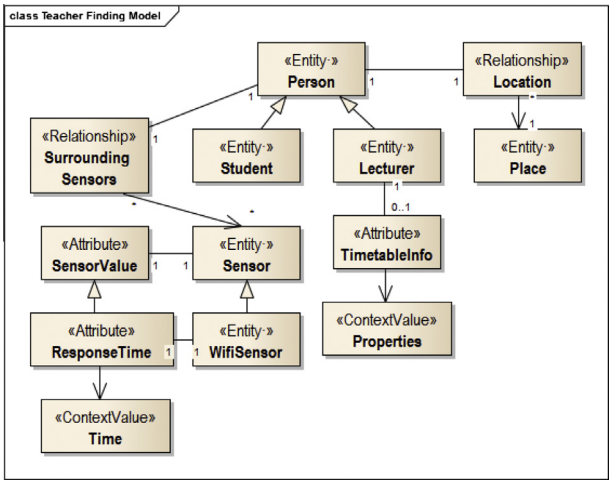

Fig. 10. Context model for the lecturer finding application.

3.4. Implementation

The *implementation* stage produces the components that implement the rules obtained during *identification* (see Section 3.2.2), for their application over the context model developed in *modeling* (see Section 3.3). These software components implement the *ContextObserver* interface. At this stage, developers need to distinguish between components that can be reused from previous applications and new ones. As will be seen, the implementation of many context observers relies on information fusion algorithms.

When the new context model extends an existing one, there may be already available *context observers* for some *ContextElements*. If these *context observers* meet the requirements of the new application, they can be used in it just launching them. For example, there can exist a component that is able to obtain the *ResponseTime* of the *WifiSensor*s connected to the *Person* by the *SurroundingSensors* relationship. Nevertheless, engineers can decide to develop new *context observers* in this case, for instance to provide alternative implementations.

The proposed set of *ContextObserver*s and their behaviors for this case study is the following. There is a *ContextObserver* component for each identified rule in Section 3.2.2:

- **GuidingApplication:** this component is started by the user of the application (typically the owner of the smartphone, the *local user*). Once started, it requests to the *context container* the *Location* attributes of both this user and the specified lecturer to make the tutorship. Its behavior consist in checking whether the *Location*s match, and otherwise, providing the user with information about how to find the lecturer.
- **SensorLocationUpdater:** this component is started as response to the first request of the *Location* attribute of the *local user* of the environment (i.e., the one carrying the smartphone). This component is unable to determine the *Location* of any other user, because it can only access to the sensors present in the node where it is running. However, the requests to it may come from a local component or a remote component via the *remote context observer*. Once started, it asks for the *SurroundingSensors* relationship in order to get the *WifiSensor*s reachable at each moment. Then, it requests the *ResponseTime* attribute for those *WifiSensor*s. Using these times, it calculates the current *Location* of the user by using previously stored knowledge. This knowledge can have different forms, such as a case base containing samples of response times and their associated locations, or a neural network likewise adjusted. Most of these alternatives would need a previous training phase. The parameters taken by this observer are $\{(\text{wfsen}_1, \mathit{RespTime}, v_1), \ldots, (\text{wfsen}_n, \mathit{RespTime}, v_n), (\mathit{user}, \mathit{Location}, \_)\}$, and it produces a context value for the relationship *Location* applied to the entity *user*.
- **SurroundingWifiSensorsUpdater:** this component is started as response to the first request of the *SurroundingSensors* relationship of the *local user* of the environment. Then, it activates the available local drivers that are able to detect Wi-fi networks. The component keeps listening to these networks and adds or removes *WifiSensor* instances as each network becomes eventually available or unavailable over time. The sensors that are not instances of *WifiSensor*, are not manipulated by this component. The parameters taken by this observer are $\{(\mathit{user}, \mathit{SurroundingWifiSensors}, \_)\}$, and it produces a context value for the relationship *SurroundingWifiSensors* applied to the entity *user*.
- **ResponseTimeUpdater:** this component is started as response to the first request of the *ResponseTime* attribute of a certain *WifiSensor*. It then starts a thread that periodically sends ping requests to the network and updates the attribute with the obtained response time. The parameters taken by this observer are: $\{(s, \mathit{ResponseTime}, \_)\}$, and produces a context value for the attribute *ResponseTime* of the sensor entity *s*.
- **TimetableLocationUpdater:** this component is started as response to the first request of the *Location* attribute of any person that is also a *Lecturer*. It is only executed in the environment representing the university, as it needs to request the *TimetableInfo* attribute in order to calculate the *Lecturer*'s location. For privacy reasons, this information is only accessible locally. The component does this task by matching the *TimetableInfo* attribute with the current local time (which is global to system and thus it is not included as a parameter). The parameters taken by this observer are: $\{(\mathit{user}, \mathit{TimetableInfo}, v_1), (\mathit{user}, \mathit{Location}, \_)\}$, and produces a context value for the relationship *Location* applied to the entity *user*, based on the *TimetableInfo* of value v_1.
- **TimetableInfoUpdater:** this component is started as response to the first request of the *TimetableInfo* attribute of a certain *Lecturer*. It then searches in its local database to obtain a *Properties* object containing all the timetable information of the specified lecturer, which is assigned to the attribute. The parameters taken by this observer are: $\{(\mathit{user}, \mathit{TimetableInfo}, \_)\}$, and produces a context value for the attribute *TimetableInfo* of the entity *user*.

The *SensorLocationUpdater* and the *TimetableLocationUpdater* constitute redundant information sources. This circumstance is automatically handled by the framework as specified in Section 2.5. Once a certain *context observer* is chosen, the other will be used only if the first becomes unavailable.

3.5. Deployment and testing

The deployment of the system includes two steps. Firstly, engineers have to run the FAERIE framework in each node of the system. In this case, there are three types of node, one for each type of environment identified. There is a university environment and two mobile environments corresponding to the smartphones of students and lecturers. Secondly, engineers have to launch the binaries of the components from the previous stages in the correspondent nodes. The binaries of the context model must exist in every node that uses the model represented in it. Those of the *context observer*s are deployed in the nodes that contain the resources they need to operate. For instance, the *context observer* that handles the lecturers' timetable information resides in the node of the university, as it locally accesses databases in that node.

4. Related work

There are numerous and heterogeneous approaches to develop context-aware systems. This is partly due to the generality of the definitions of *context* and *context-awareness*, which makes that very different applications can claim being context-aware, and the wide range of mechanisms, technologies and requirements appearing in such systems.

As a general overview, the presented work represents a simpler approach than other more detailed and formal general alternatives, such as Sánchez-Pi et al.' work [15]. On the other hand, works such as Turaga et al.' [16], offer simple and efficient architectures to develop domain-specific systems (e.g., surveillance systems). However, these architectures are conceived to be ad hoc, which may harm interoperability with other context-aware systems. In a middle way, FAERIE offers a well-balanced trade-off between simplicity and generality.

This section tries to cover the broad landscape of features that distinguish architectures through the following subsections. Each one discusses some of the main characteristics commonly used to evaluate context-aware solutions. Table 1 summarizes their results.

4.1. Context acquisition

Context-aware systems are embedded in the environments that they need to know. However, most of their components do not directly access the sensors, but delegate this task to some mediator. This mediator organizes the access to the information in order to guarantee, for instance, fairness and security for requests. The kind of mechanism used for this *context acquisition* constitutes the first analysis dimension. There are two main types of context acquisition: *interface* and *data-oriented*.

In *interface-oriented* acquisition, the client component gets an interface to some kind of component and uses it to ask and get information. This can be done by using widgets or context servers.

A *widget* is a software component that provides a public local interface to a hardware sensor. This is the approach of the *Context Toolkit* [6]. The widget does not implement functionality for remote access, so it needs additional components to be used in distributed environments as those proposed in FAERIE. As an advantage, it provides an efficient use (i.e., with a low management overload) as long as no concurrent access is needed.

A *context server* aggregates the services of multiple context providers [22,23,25]. Context consumers request to such context server a service able to provide the context information they need. When they get the service, they interact with it to get the information. A variant of this mechanism is a *context broker*, which plays the same role but gives access to provider agents instead of services [24]. This approach can hide the differences between local and remote information, as FAERIE does, but it also brings some additional processing. While FAERIE clients only need to process the data, those of context servers also have to manipulate the service.

The other main approach for context acquisition is *data-oriented* mechanisms. They rely on a central repository of information where components access and request specific data. *Blackboard models* belong to this category.

A blackboard provides an indirect context acquisition: a component makes a declarative request to the blackboard, generating an opportunistic coordination of components to provide that information, which is put accessible for the client component in the blackboard when it is available. The use of this model decouples the context providers from the context consumers, facilitating system reconfiguration. As a consequence, the system is more robust, as long as the context server stays up. *iRoom* [5] is one of the first attempts to apply the blackboard model to this field. Haya et al. [19] present a similar work, enhanced with the possibility to access the context by navigating through the relationships in the context model graph. The main drawback of this approach is efficiency, as every communication in the system must pass through the context server. The *Open Context Platform* [20] addresses this issue. It can work without the existence of a central context server, making use of a tuple space that conforms a *virtual blackboard*. The tuple space assigns a unique identifier to each context element. This identifier is also used as the address to find the context handler that manages that information. In this way, when a component requests a certain context element, the request is redirected to the correspondent component, which provides the value directly. This platform can also work using a central server. In any case, if for some reason the handler component is not found, the request is redirected to an existing server.

Another key difference between the *interface-oriented* and *data-oriented* approaches is the location of the mechanism to maintain consistency. In the first case, the client chooses among the different interfaces, and thus it has the responsibility to deal with redundancy and inconsistency. For that task, it can make use of domain-specific heuristics. In the second case, since the client does not usually choose the context provider, the responsibility to deal with redundancy and inconsistency lies on the framework. This relieves the client of this management but, in turn, it may makes that management less efficient, because solutions tend to be more general.

4.2. Context modeling

Once that context has been acquired, the system needs to represent it in a suitable way for its functionality. How to do it constitutes the second characteristic, *context modeling*. There are different ways of implementing the context model [9]: ontology-based, object-oriented, and key-value tuples. These approaches are sometimes enhanced with markup schemes or logic propositions to provide additional semantics to the model. Some frameworks also provide graphical means to specify the context types. Henricksen et al. [21] present one as an extension of the object-relational mapping, and Venturini et al. [24] integrate the context model with the Belief-Desire-Intention (BDI) [26] model from software agent theory.

The simplest representation of the context is possibly as a set of key-value tuples. A single context element is represented as a list of numeric and string values, each one labeled with a key to indicate the type of information it represents. The *Context Toolkit* [6] is an example of this approach. The main advantage of this representation is its simplicity, both for use and extension; however, that simplicity implies limited facilities to work with the context information.

Ontology-based approaches are the most popular because of its expressiveness and extensibility. *CoBrA* [27], the *Open Context Platform* [20], and the works of Gu et al. [28], Ejigu et al. [22], and Venturini et al. [24] propose ontology-based modeling for context information. They require integrating an ontology parser and engine to use ontologies, thus demanding higher resources than other approaches. Moreover, such powerful capabilities are not required in many systems, although they could be desirable. For this reason, numerous works adopt some of the other simpler alternatives.

The object-oriented approach for context modeling is also widely used, for instance in the *Java Context-Aware Framework* (JCAF) [29], the *Hermes* project [23], and FAERIE as well. This approach allows leveraging existing technologies for object-oriented languages, such as object serialization or mapping to relational databases for storage. Thus, it brings important savings in formation for development teams and tool acquisition, and reduces the costs related with the heterogeneity of technologies in projects.

4.3. Context processing

Raw information, as directly obtained from sensors, is rarely used by context-aware components. This information usually goes through fusion (aggregation or interpretation) processes to generate a representation in terms of the application domain. This is done following either a *data-centric* or a *process-centric* approach [4].

The *data-centric* approach specifies transformations in terms of the data to process. This specification largely isolates the transformation process from changes in the components that effectively perform them. A popular way to implement this approach is through declarative transformation rules. This kind of rules speci-

fies preconditions and postconditions on the context representation. When the context matches the required precondition, the rule is triggered to establish the specified postcondition. The kind of aggregation that can be done in this way is limited by the expressive power of the language for the rules. The *Open Context Platform* [20] uses this approach. Other works adopt an imperative implementation: the elements processing the context contain the necessary instructions to check the context conditions and perform the required changes. This last solution does not need an additional rule interpreter as the declarative one does, but in turn its code is less human-readable. *iRoom* [5], the *Context Management Framework* [17], Haya et al.' work [19], and FAERIE follow this approach.

In the *process-oriented* approach, the aggregation is specified in terms of the context providers. This gives developers more control about how to deal with redundant sources of context information, as they specify their composition. However, this approach needs additional adaptations when these sources change, something that does not happen with data-centric approaches. This approach is used in the *Context Toolkit* [6], the works of Henricksen et al. [21] and Ejigu et al. [22], and the *Hermes* project [23]. A notable example of this approach is Venturini et al.' work [24], which defines the context processing in terms of agent protocol diagrams that specify the different outcomes of context exchange between agents.

4.4. Distribution and layering

Other characteristics to analyze are the framework *distribution* and *layering*. The *distribution* refers to the way the different elements of a system are placed in computational devices, and it distinguishes whether the framework uses central servers or not. The *layering* considers the organization of the components of the system, distinguishing between *static* and *dynamic* organizations.

The use of a centralized server relieves distributed components of the burden of context processing, and facilitates maintaining the consistency of the context representation using general (i.e., not domain-specific) mechanisms. However, it limits the scalability of the system and makes it less fault-tolerant. *iRoom* [5], *Hydrogen* [18], the *Context Management Framework* [17], and Haya et al.' work [19] follow this approach. When there is no centralized server, nodes need to perform additional processing in order to keep consistency, but the whole system scales better. FAERIE adopts this approach, as the *Context Toolkit* [6], OCP [20], the *Hermes* project [23], and the works of Henricksen et al. [21], Ejigu et al. [22], and Venturini et al. [24].

Regarding the organization of components, the *static* layering means that abstraction layers are predetermined by the framework. This makes systems more uniform and facilitates the detection of errors. The *Context Toolkit* [6], *Hydrogen* [18], Henricksen et al.' work [21], OCP [20] and Ejigu et al.' work [22] use this approach. On the other side, the *dynamic* layering implies that abstraction layers appear in the system at runtime. The resulting systems gain in flexibility to establish at which level of abstraction to work. However, this adaptive structure also makes more difficult for engineers to detect errors in the system, as they can make fewer assumptions about the current state and structure of the system. This is the approach in *iRoom* [5], the *Context Management Framework* [17], Haya et al.' work [19], the *Hermes* project [23], Venturini et al.' [24], and FAERIE.

5. Conclusions

This paper has described the architecture and use of FAERIE, a framework for context-aware systems. It focuses on the management of information (i.e., update and use) following a distributed variant of the blackboard model. Its specification relies on three main types of components: *context elements*, *context containers*, and *context observers*.

The set of *context elements* represents the "context model" with an object-oriented approach. This choice offers a good trade-off between performance, extensibility and reutilization with respect to ontology-based and key-value approaches. Performance is high thanks to a tight integration with processing code. Extensibility and reutilization are facilitated by the use of mainstream object-oriented mechanisms. This also facilitates the framework adoption, as its modeling paradigm is common background and practice for engineers.

Context containers and *context observers* manipulate the previous elements. Containers bind suitable observers with the *context elements* they request to use or are able to update. Containers are transparently federated through *remote context observers*, thus producing a distributed virtual blackboard. This distribution makes systems more robust and scalable than those using a central blackboard server. Also, it is more efficient in terms of network usage than those without containers, since each single environment is able to work with its local context container as long as it does not need any information from other environments.

This architecture supports opportunistic data-centric coordination between components. The data-centric approach improves transparency about the specific processing components that a system includes, and decouples context acquisition and use. This is because the coordination is specified in terms of the data being processed. The alternative would need to specify which components of the system are aggregated, thus coupling it to a specific structure. These features simplify the management of the framework, as it is easy to modify the involved components, and improves its robustness, as there can be redundant observers for the different tasks, even in different nodes. The opportunistic processing means that each component only starts its processing when another component has requested the information it can handle. This saves resources when there is no component interested in a certain property (e.g., a sensor can turn into stand-by if there is no component interested in its lecture). This differs from other blackboard approaches that create "context channels" where context information is constantly provided. In addition, the framework automatically manages the reconfiguration of the information flows, for instance when new components register in a node or old components become unavailable.

Finally, the architecture provides flexibility for every observer component to work at the needed level of abstraction, as it does not define static layers. For example, a component may access both a property provided directly by a sensor and another calculated through procedures of aggregation, interpretation and reasoning. This facilitates rapid prototyping and incremental development of applications, while other solutions need to establish many abstraction layers to have a complete running application. Also, the use of the framework facilitates the development in the sense that it hides the details that are not related with the business logic of the application. This leaves two main tasks: definition of the context structure and its changes. The structure is obtained through the analysis of the problem, which may include extending existing context models. The context changes are specified indicating which processes (including fusion) are applied to each *ContextElement* type.

The presented work is part of a wider effort to provide a general architecture and infrastructure for AmI applications. In this context, there are still several open issues. First, the context model needs to support the temporal dimension of information. In its current state, FAERIE calculates the context on time $t+1$ using the context on time t. The objective would be to be able to calculate the context on time $t+n$ using the context on times $t, t+1, \ldots, t+n-1$. FAERIE could incorporate these requirements

by implementing an extra *context observer*. It would be responsible for registering all changes in the context and their timestamps, and providing access to a representation of them. This would allow, for instance, systems learning from past experiences to improve their performance. Second, there are privacy issues when dealing with personal data. Since each node owns its private context, it would be possible to implement policies in their *remote context observers* in order to regulate which information can be remotely requested. Third, it has been mentioned that the use of distributed blackboards gives an opportunity to improve robustness and performance. The framework can implement mechanisms to replicate information among different nodes, providing alternative points of access to certain information. More improvements in performance can be made regarding the memory usage of the context container. It can be limited with configuration parameters, and the container can save memory transferring the information that is not frequently used from memory to permanent storage. Finally, the inclusion of real-time requirements would support new scenarios, such as control of manufacturing installations or medical centers.

Acknowledgments

This work has been done in the context of the project "Social Ambient Assisting Living – Methods (SociAAL)", supported by the Spanish Ministry for Economy and Competitiveness, with Grant TIN2011-28335-C02-01. Also, we acknowledge support from the "Programa de Creación y Consolidación de Grupos de Investigación" UCM-BSCH GR35/10-A.

References

[1] P. Remagnino, H. Hagras, N. Monekosso, S. Velastin, Ambient intelligence, in: P. Remagnino, G. Foresti, T. Ellis (Eds.), Ambient Intelligence, Springer, New York, 2005, pp. 1–14.
[2] G. Abowd, A. Dey, P. Brown, N. Davies, M. Smith, P. Steggles, Towards a better understanding of context and context-awareness, in: H.-W. Gellersen (Ed.), Handheld and Ubiquitous Computing, Springer, Berlin, 1999, pp. 304–307.
[3] M.B.A. Haghighat, A. Aghagolzadeh, H. Seyedarabi, Multi-focus image fusion for visual sensor networks in DCT domain, Comput. Electr. Eng. 37 (2011) 789–797.
[4] M. Baldauf, S. Dustdar, F. Rosenberg, A survey on context-aware systems, Int. J. Ad Hoc Ubiquit. Comput. 2 (2007) 263–277.
[5] T. Winograd, Architectures for context, Hum.–Comput. Interact. 16 (2001) 401–419.
[6] A.K. Dey, G.D. Abowd, D. Salber, A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, Hum.–Comput. Interact. 16 (2001) 97–166.
[7] K.E. Kjær, A survey of context-aware middleware, in: W. Hasselbring (Ed.), Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering (SE 2007), ACTA Press, Innsbruck, 2007, pp. 148–155.
[8] R. Schmohl, U. Baumgarten, Context-aware computing: a survey preparing a generalized approach, in: S.I. Ao, O. Castillo, C. Douglas, D.D. Feng, J.-A. Lee (Eds.), Proceedings of the International MultiConference of Engineers and Computer Scientists (IMECS 2008), Newswood Limited, Hong-Kong, 2008, pp. 744–750.
[9] T. Strang, C. Linnhoff-Popien, A context modeling survey, in: N. Davies, E.D. Mynatt, I. Siio (Eds.), Proceedings of the Workshop on Advanced Context Modelling Reasoning and Management at UbiComp 2004, Springer, Nottingham, 2004, pp. 1–8.
[10] D. Steinberg, S. Cheshire, Zero Configuration Networking: The Definitive Guide, O'Reilly Media Inc., Sebastopol, 2005.
[11] J. Rellermeyer, G. Alonso, T. Roscoe, R-osgi: distributed applications through software modularization, in: R. Cerqueira, R. Campbell (Eds.), Middleware 2007, Springer, Berlin, 2007, pp. 1–20.
[12] M. West, J. Harrison, Bayesian Forecasting and Dynamic Models, second ed., Springer-Verlag New York, Inc., New York, NY, USA, 1997.
[13] M.M. Kokar, C.J. Matheus, K. Baclawski, Ontology-based situation awareness, Inform. Fusion 10 (2009) 83–98.
[14] D.S. Taubman, M.W. Marcellin, JPEG 2000: Image Compression Fundamentals, Standards and Practice, Kluwer Academic Publishers, Norwell, MA, USA, 2001.
[15] N. Sánchez-Pi, J. Carbó, J.M. Molina, A knowledge-based system approach for a context-aware system, Knowl.-Based Syst. 27 (2012) 1–17.
[16] P. Turaga, Y. Ivanov, Diamond sentry: integrating sensors and cameras for real-time monitoring of indoor spaces, IEEE Sens. J. 11 (2011) 593–602.
[17] P. Korpipaa, J. Mantyjarvi, J. Kela, H. Keranen, E.-J. Malm, Managing context information in mobile devices, IEEE Pervasive Comput. 2 (2003) 42–51.
[18] T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, J. Altmann, W. Retschitzegger, Context-awareness on mobile devices - the hydrogen approach, in: W.P. Company (Ed.), Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS 2003), IEEE Computer Society, Washington, 2003, pp. 292–301.
[19] P.A. Haya, A. Esquivel, G. Montoro, M. García-herranz, X. Alamán, R. Hervás, J. Bravo, A prototype of context awareness architecture for ambience intelligence at home, in: M. Research (Ed.), Proceedings of the International Symposium on Intelligent Environments 2006, Microsoft Research Ltd., Cambridge, 2006, pp. 49–55.
[20] I. Nieto, J.A. Botía, A.F. Gómez-Skarmeta, Information and hybrid architecture model of the OCP contextual information management system, J. Univ. Comput. Sci. 12 (2006) 357–366.
[21] K. Henricksen, J. Indulska, Developing context-aware pervasive computing applications: models and approach, Pervasive Mob. Comput. 2 (2006) 37–64.
[22] D. Ejigu, M. Scuturici, L. Brunie, Hybrid approach to collaborative context-aware service platform for pervasive computing, J. Comput. 3 (2008) 40–50.
[23] S. Buthpitiya, F. Luqman, M. Griss, B. Xing, A. Dey, Hermes – a context-aware application development framework and toolkit for the mobile environment, in: L. Barolli, T. Enokido, F. Xhafa, M. Takizawa (Eds.), Proceedings of the 26th International Conference on Advanced Information Networking and Applications Workshops (WAINA 2012), IEEE Computer Society, Fukuoka, 2012, pp. 663–670.
[24] V. Venturini, J. Carbó, J.M. Molina, Methodological design and comparative evaluation of a MAS providing AMI, Expert Syst. Appl. 39 (2012) 10656–10673.
[25] J.I. Hong, J.A. Landay, An infrastructure approach to context-aware computing, Hum.–Comput. Interact. 16 (2001) 287–303.
[26] M.E. Bratman, Intention, Plans, and Practical Reason, CSLI Publications, New York, 1999.
[27] H. Chen, T. Finin, A. Joshi, An ontology for context-aware pervasive computing environments, Knowl. Eng. Rev. 18 (2003) 197–207.
[28] T. Gu, H.K. Pung, D.Q. Zhang, A service-oriented middleware for building context-aware services, J. Netw. Comput. Appl. 28 (2005) 1–18.
[29] J. Bardram, The java context awareness framework (jcaf) – a service infrastructure and programming framework for context-aware applications, in: H. Gellersen, R. Want, A. Schmidt (Eds.), Pervasive Computing, Springer, Berlin, 2005, pp. 98–115.