An Architecture for the Design of Platforms Supporting Responsive Environments

Marco Covelli[1], Daniela Micucci[2] and Marco Mobilio[2]

[1] *TabulaeX, via Carducci 32, Milan, Italy*

[2] *Department of Informatics Systems and Communication, University of Milano Bicocca, Viale Sarca 336, Milan, Italy*

Keywords: Perception Flow, Action Flow, Integration, Software Architecture, Responsive Environments

Abstract: Responsive environments are able to sense the environment and to respond to it and to the users that inhabit it. Those systems require both the integration of heterogeneous devices and an abstract representation of the environment to reason about interesting changes. The paper presents DEA (Domain Entities Access), an architecture that enables the realization of platforms supporting responsive environments in the interaction with instrumented physical environments through the observation and the control of meaningful domain entities, thus abstracting from any technological details. Platforms can be easily realized by plugging specific domain-dependant components in a framework that manages all the domain-independent aspects. Thus, the architecture results to be open with respect to both new devices and new typologies of domain entities. A prototypical implementation of the framework has been provided. Moreover, a specific platform has been realized to support an end-user application dealing with instrumented environments.

1 INTRODUCTION

Instrumented environments (Butz and Krüger, 2003) are common environments enriched with devices able to gather information about them and to act on them. From a technological point of view, they constitute the milestone of the responsive environments (Negroponte, 1975; Bullivant, 2006), systems able to sense the environment and to respond to it and to the users that inhabit it.

Those kind of systems primarily require to intermix multiple components and integrated solutions (e.g., home automation gateways) that are highly heterogeneous, have different capabilities, and often rely on different communication protocols (Kim et al., 2012). Due to this heterogeneity, many systems rely on *ad hoc* solutions that often are based on specific technologies and protocols.

The approaches to the integration of heterogeneous devices can be divided into two main groups: solutions that supply with enabling integration platforms (Thomson et al., 2008; Kusznir and Cook, 2010; Ristau, 2008), and solutions that provide platforms that allow applications to reason in terms of domain-related concepts (Román et al., 2002; Aiello and Dustdar, 2008). Platforms of the first group provide an unified access to the heterogeneous devices. Thus, they can be used in any application domain, but they do not provide an abstract, domain-dependant model of the environment. On the opposite, solutions of the second group provide applications with a model of the environment that is closer to the application logic, thus filling the gap between the physical environment and how it is perceived by the applications. The main disadvantage of such solutions concerns the poor adaptability of the model to different domains than the original.

The paper proposes Domain Entities Access (DEA), an architecture for the observation and the control of instrumented environments that is located halfway between the two main classes of approaches. DEA allows the integration of heterogeneous devices and provides end-user applications with a unified access to an abstract domain-related representation of the environment. The abstract representation of the environment i) captures domain related issues by abstracting from the physical devices; ii) can be inspected by end-user applications with the aim of identifying intelligent/ad hoc behavior; and iii) can be used by end-user applications to deliver commands reifying the identified intelligent/ad hoc behavior.

The architecture carefully separate issues related to the specific application domain (e.g., inferring the position of a person from stimuli generated by a cam-

Covelli M., Micucci D. and Mobilio M..
An Architecture for the Design of Platforms Supporting Responsive Environments.
DOI: 10.5220/0005099404170427
In *Proceedings of the 9th International Conference on Software Engineering and Applications* (ICSOFT-EA-2014), pages 417-427
ISBN: 978-989-758-036-9

era) from those that are domain-independent (e.g., the way applications observe and control the environment). Moreover, the architecture enforces the insulation and compactness of components, and, in particular, of those that are domain-dependant. Such an approach allows defining a framework that provides both an implementation of all the domain-independent and the infrastructure in which easily plugging the domain-dependant components. This results in a platform that end-user applications can exploit to observe and control domain-related entities.

The paper is organized as follows: Section 2 presents the DEA architecture; Section 3 describes the implementation of a framework supporting the DEA architecture and a specific configuration for a real simplified scenario; Section 4 compares DEA to the state of the art; and Section 5 outlines the conclusions and identifies future directions.

2 DEA ARCHITECTURE

There is a semantic gap between the *environment model* used by end-user applications to observe and interact with the physical environment and the devices that produce *stimuli* and actuate *actions*. End-user applications reason in terms of *statuses of domain entities*. For example, "*Marco* is *located* in *room 27*", "switch on the *main light* in *room 2006*". All the emphasized words are domain entities and related statuses. On the opposite, sensing devices produce *stimuli* and actuating devices accept *actions* whose semantics and syntax is up to the devices. "DF6YH78KLO", "#01001#01", are respectively examples of a stimulus from a RFID reader and of an action to a BTicino light.

DEA (Domain Entity Access) is a layered architecture for the design of platforms supporting end-user applications that reason in terms of domain entities be they abstractions of physical devices (e.g., lamp) or inferred from events generated by sensing devices (e.g., people), thus filling the semantic gap. The architecture seamlessly integrates sensing and actuation devices, providing end-user applications with a *environment model* that they can exploit to control and observe the *status* of meaningful *domain entities*. The environment model is an abstract and unified representation of the *context of interest*, which ranges from the physical devices (e.g., lamps) to the people that inhabit the environment.

2.1 Overview

Stimuli from the sensing devices in the physical environment contribute in maintaining the environment model updated so that it can reflect the "real" situation. Symmetrically, *commands* from end-user applications possibly affect the "real" environment through *actions* that are performed by actuating devices. In turn, a change of the "real" environment is captured by sensing devices that produce stimuli, thus closing the loop. For example, an application that tracks people and activates cameras only when required, reasons on a model of the environment constituted by people and cameras whose status is updated by a set of physical cameras, RFID readers, and any kind of sensing device able to detect movements. Moreover, the application operates on the status of the camera in the environment model to control the corresponding physical camera, thus ignoring the specific technological dependant action required to switch on/off the physical camera. In turn, when the physical camera changes its status, the corresponding generated stimulus will update the status of the camera in the environment model. The two flows respectively realize the processes of perception and action described in (Cook and Das, 2007).

Referring to Figure 1, the first three layers of the architecture (from the bottom) deal with *data abstraction* that is responsible for maintaining the environment model updated with respect to the "real" environment, thus managing both the *perception* and the *action* flows. The upper layer deals with *access mechanisms* end-user applications can use to observe and control the environment model. In detail, the *interface layer* is responsible for interfacing with the specific device; the *translation layer* translates stimuli as produced by the devices into a common vocabulary (abstract stimuli) and actions (abstract actions) into technological dependant actions; the *inference/reification layer* makes inferences about statuses of domain entities according to stimuli from the devices (and the actual statuses) and reifies commands into abstract actions (independent from any technological issues) that actuators has to perform; finally, the *access layer* provides mechanisms an end-user application can exploit to observe and control statuses of domain entities.

2.2 Data Abstraction

The two lowest layers of the architecture concern the interfacing with a set of heterogeneous devices, each of them communicating through its own protocol and producing or consuming data according to its own

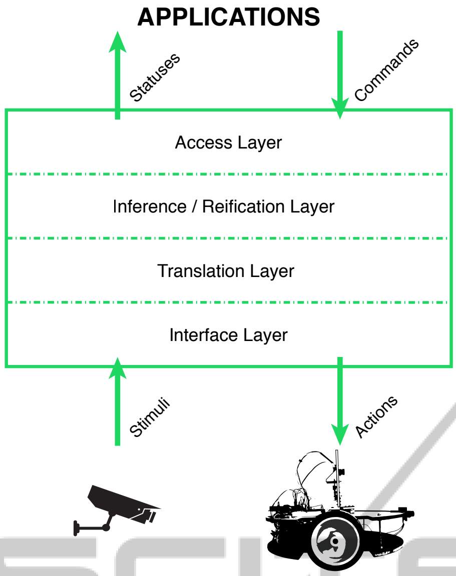

Figure 1: Overall architecture.

specific syntax.

The interface layer handles the specific communication mechanisms with sensors and actuators. Respectively i) it receives *stimuli* from sensing devices and delivers them to the translation layer, and ii) it accepts *actions* from the translation layer and delivers them to the correct actuating devices.

At this layer, stimuli are still related to a specific syntax: the translation layer abstracts from the specific syntax of stimuli, allowing the higher layer to focus on their semantics only. Respectively, i) it receives stimuli from the interface layer, turns their syntax to a common language maintaining the original semantics by generating *abstract stimuli*, and delivers the abstract stimuli to the inference/reifying layer; ii) it receives abstract actions (i.e., actions expressed in the common language) from the inference/reifying layer, generates actions by turning the abstract actions in the specific syntax known by the actuators recipients of the actions, and delivers the actions to the interface layer. This way, the inference/reifying layer can rely on data (abstract stimuli and abstract actions) that are completely independent from their specific devices. For example, in this layer two stimuli of brightness respectively from a PWM (pulse-width modulation) and a serial interface, will be standardized to a common scale and syntax.

Stimuli and actions are completely independent from their specific devices. The inference/reification layer relies on those information to infer the *statuses* of *domain entities* and to reify *commands* into actions.

Domain entities ("entities" from now on) realize the environment model. They are observable and possibly controllable units of interest in a "real" environment from the application point of view. An entity is defined as a set of *property-value* pairs, that entirely describes the entity itself. Each property models a piece of information. Which properties characterize an entity is a domain related issue. For this reason, DEA specifies only how they have to be defined. Referring to a domotic domain, Figure 2 sketches two examples of domain entities that respectively represent a person and a light each characterized by its proper set of properties: a person has a position and a name, a light has a position too and is in an on/off status. Moreover, both the entities has a type and an unique identifier.

Figure 2: Examples of domain entities.

Properties can be mutable and immutable. The former are fixed and cannot be changed over time (e.g., the name of a person). On the opposite, the values of mutable properties are the results of a inference process operated in the inference/reification layer. The process evaluates abstract stimuli from the translator layer to infer significant changes of domain entities properties. In real cases, abstract stimuli could be inaccurate, mostly due to sensors quality or intrinsic difficulty of the perception task. For this reason, the inferred values are enriched with *confidences*. Thus, stimuli from the field activate perception flows that cause updates of the statuses of domain entities.

Commands are requests for changing the status of entities. This means that a status can also be controllable. It is important to notice that not all statuses can be also controllable: although the on/off status of a light is typically controllable, the same cannot be asserted for a person's location. When a commands is delivered to the inference/reification layer, it is reified into abstract actions that are then delivered to the translator layer. Thus, commands activate action flows that cause the activation of the physical actuators.

Figure 3 sketches an example of the perception flow. An RFID sensor detects a tag: this event is captured by the interface layer that exposes the data to the upper layer. The data is translated into the homogeneous syntax and propagated to upper layer. The inference/reification layer infers the new value for the position property of the person with identifier `m_covelli` and then updates the persistent representation of the environment model.

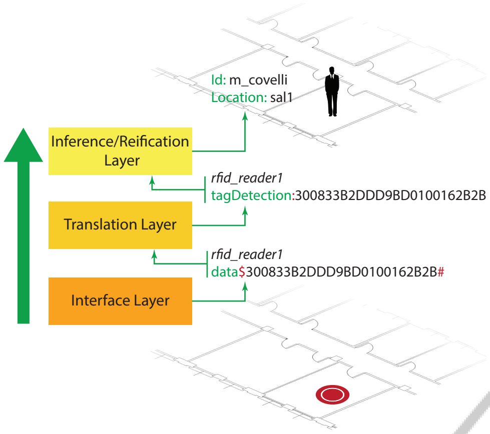

Figure 3: Perception flow.

Figure 4 sketches an example of the action flow. An end-user application wishes to turn a light on. The

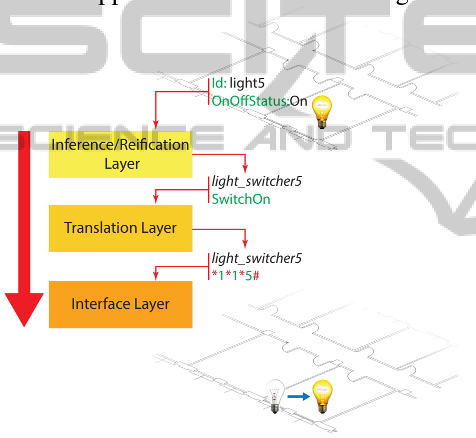

Figure 4: Action flow.

light is modeled by a domain entity with a set of properties including `OnOffStatus`, which is controllable and contains the status of the light. The application delivers to the inference/reification layer a command stating that the value of the property `OnOffStatus` should be set to `On`. The layer is in charge of reifying the command by producing the proper abstract action for the corresponding light switcher. The abstract action is delivered to the translator layer that produces an action that is understandable to the recipient light switcher. The action is then managed by the interface layer that finally deals with the concrete interfacing with the device.

2.3 Environment Model Access

The inference/reification layer maintains entities (i.e., the environment model). The access layer provides mechanisms end-user applications can exploit to observe and control the environment model. Such mechanisms are based on messages and allow formulating requests about domain entities without the need to mention them explicitly. By exploiting a subset of the concepts of predicate logic, it is possible to refer to domain entities through their properties and their values. Such a solution allows end-user applications to do not explicitly know the domain entities constituting the environment model. For example, an application can formulate a request like “switch on the lamp in room 27” without explicitly knowing which is the lamp in room 27. At a conceptual level, the approach is to send messages directly to domain entities, which respond individually on their merits. Reply messages are also characterized by a payload that contains the required information, and by a sender that identifies the entity to which the information is referred.

A *request message* consists of a *recipient*, which describes via predicate logic the properties of the entities to which the message is addressed, and a *payload*, which specifies the detail of the request. A *reply message* consists of a *sender* that is described via predicate logic and a *payload* with the information related to the sender.

We define a predicate $p(x)$, with x a domain entity, as a series of property-value couples, linked by the common logical connectors (conjunction, disjunction, and negation). For example, the predicate $p_1(x)$ “x is a lamp located in sal2 lab” can be expressed in terms of property-value tokens like “x has property `Type` equals to `Lamp` AND x has property `Location` equals to `sal2`”. Defining the environment model E as the set of all the domain entities and a given $p(x)$, it is possible to declaratively describe a set $E_p \subseteq E$, containing the entities having the characteristics described in p, as follows: $E_p = \{e \in E \mid p(e) \text{ is true}\}$. To be fully compliant with the domain model, the syntax includes the possibility to specify a minimum confidence, to filter values under a given trustworthiness threshold.

This approach effectively allows entity selection by the specification of property constraints.

The use of predicate enables the definition of dynamic sets of entities, by formalizing predicates that may include mutable property values.

For example, it is possible to define a predicate $p_2(x)$ “x is a person in sal2 lab”: there will be a concrete possibility that an entity e could be in E_p at the moment t_0 but not at t_1. This makes possible to discriminate entities by their properties, without the need of enumerating them.

However, it is important to notice that this approach also fits the case we want to explicitly refer to a specific entity, that could be done by defining a constraint on the `Id` property (if defined).

Exploiting the above described message-based protocol, the access layer enables end-user applications

- to query the model about the punctual status of selected entities (*observation*)
- to express interest for statuses' changes of selected entities, obtaining notifications at each occurrence (*subscription*)
- to express desired statuses for entities, that are reified in changes to the physical environment made by suitable actuators (*wish*)

Observation and subscription are for observation purposes. However, in the first case the request concerns the current status of an entity; in the second one, it refers to the status changes that occurs since the request. Instead, wish allows end-user applications to deliver commands, i.e., to change the entities status. For example, observation allows to query the environment in order to obtain the names of the persons in a room at the time of the request; subscription allows to express interest for all the future changes of the status of the lights in a specific room, without having to explicitly list them; wish allows to ask for switching off all the lights in a certain area.

An *observation* consists in a request message specifying a predicate that defines the interested entities and a list L of properties and corresponding values as payload.

For example, an end-user application needs to know the number of people that are currently present in a building composed by two rooms. `Room` is a domain entity characterized by the properties `Id` (the identifier), `Type` (the typology of the entity), and `ContainedPeolple` (the number of contained people). Thus, the end-user application composes the observation request message:

```
Recipient:
   Type = Room
ObservationRequest:
   ContainedPeople
```

This message is delivered to all the entities who have the property `Type` equals to `Room`. The payload specifies that the request concerns the value of their property `ContainedPeople`. The two rooms (`sal1` and `sal2`) *answer* the query by sending back to the requesting application, the messages:

```
Sender:
   Id = sal1
ObservationResponse:
   ContainedPeople = 1 0.9

Sender:
   Id = sal2
ObservationResponse:
   ContainedPeople = 3 0.9
```

The second value in `ContainedPeople = 1 0.9` and in `ContainedPeople = 3 0.9` is the confidence value.

Subscription allows to observe the environment model asynchronously: end-user applications subscribe to entity status changes so that they will be notified each time a change occurs. Firstly the end-user application performs a subscription specifying the predicate p that describes the target entities and the list L of properties in which it is interested. Since the subscription, the end-user application will receive a notification whenever a status change involves one of the properties in L of an entity in E_p.

For example, an end-user application needs to be notified each time a student changes its location inside a university building. `Person` is a domain entity characterized by the properties `Id` (the identifier), `Type` (the typology of the entity), `Location` (the position inside the building), and `Role` (the role of the person). Thus, the end-user application composes the subscription request message:

```
Sender:
   CaseStudyApp
Recipient:
   Type = Person AND
   Role = Student
SubscriptionRequest:
   Location
```

From now on, the requesting application will be notified of any `Location` value change that involves entities of type `Person` and role `Student`. Differently from "standard" request messages, the subscription also includes a *sender* field, needed to identify the recipient of future notification messages. Suppose that the environment model has been updated as a result of a perception flow generated by an image captured by a camera. The `Location` value of the entity with `Id` equals to `m_covelli` has changed, meaning that the entity "has entered" a new location. Then, the entity itself sends to the applications that are subscribed to such event the notification message:

```
Sender:
   Id = m_covelli
StatusChange:
   Location = sal1 0.8
```

Wish allows end-user applications to control the "real" environment through commands. Actuated commands can produce effects that are perceived by sensors, which activate a perception flow. Thus, the consequences of a command request will be observable by the applications if they properly observe the environment model, according to the previous introduced modes (observation and/or subscription). In other words, to perceive the change, the application must observe the entity it wants to control.

A wish consists in a request message that contains a predicate p that describes the target entities, and a *property-value* pair as payload, that specifies the property and the new value the application "wishes" to assign to the target entities.

For example, an end-user application needs to switch on all the lights in the `sal1` room. `Light` is a domain entity characterized by the properties `Id` (the identifier), `Type` (the typology of the entity), `Location` (the position inside the building), and `OnOffStatus` (the on/off status). Thus, the end-user application composes the request message:

```
Recipient:
   Type = Light AND
   Location = sal1
WishRequest:
   OnOffStatus = On
```

This request message selects the entities by the type (they should be lights) and the location (they should be in sal1), and asks them to switch on. This request activates an action flow.

2.4 Concrete Architecture: Components

Figure 5 illustrates the overall architecture with emphasis on its concrete realization in terms of software components.

Each of the first three layers deals with well-defined data structures both in perception flow (from bottom to top) and in action flow (from top to bottom). This allows identifying software components characterized by compactness and insulation (Stevens et al., 1979).

In detail and starting from the bottom, the component in charge of communicating with a device is the *sensor wrapper* (for sensing devices) and the *actuator wrapper* (for actuating devices). At least there are as many wrappers as the different typologies of the physical devices. In Figure 5 they are represented by the components labeled $\mathtt{SW}_i$ (sensor wrappers) and $\mathtt{AW}_i$ (actuator wrappers).

The component in charge of operating translations is the *stimuli translator* (from stimuli to abstract stimuli) and the *action translator* (from abstract actions to actions) respectively. At least there are as many translators as the different typologies of protocols used by the physical devices. In Figure 5 components labeled $\mathtt{ST}_i$ and $\mathtt{AT}_i$ are respectively stimuli translators and action translators.

Wrappers and translators depend on the specific devices that instrument the environment. Thus, they are domain-dependant components.

The inference and the reification activities in the inference/reification layer are respectively concretized by the *status guesser* and the *wish reasoner* components.

In Figure 5, components labeled $\mathtt{SG}_i$ and $\mathtt{WR}_i$ are respectively status guessers and wish reasoners. How many guessers are needed depends both on the characteristics of the domain entities (i.e., their properties and dependencies) and on how much the guessers are compact and insulate. The same holds for the wish reasoners.

Guessers and reasoners depends on the specific domain entities that constitute the environment model and their properties. Thus, they are domain-dependant components.

Each interaction mode is supported by specific components as depicted in Figure 5: the *observation* component is in charge of managing the observation interaction mode; the *subscription* component is in charge of managing the subscription interaction mode by capturing the status changes inferred by the status guessers and delivering them to the subscribed end-user applications; and the *wish* component is in charge of managing the status change requests, thus delivering them to the proper wish reasoners.

The identified components and layering allow to define a *framework* for what concerns the access layer and the structure of the components in the data abstraction layer. When an instrumented environment must be observed and controlled, then a platform is designed. Such a *platform* will relies on the framework for what concerns the domain-independent issues, and will include both the appropriate set of domain entities and the domain-dependant components.

3 VALIDATION

The validation aimed to prove the effective advantages in the development of end-user applications using the presented architecture. The validation process includes the concrete design and the implementation of:

- *DEA framework*, which provides the concrete implementation of all the domain-independent components, the interface of the domain-dependant components, and the components interaction mechanisms.
- *DEA platform*, which includes the definition of the entities typologies involved in a specific domain and the implementation of all the abstraction layer components according to the interfaces provided by the DEA framework

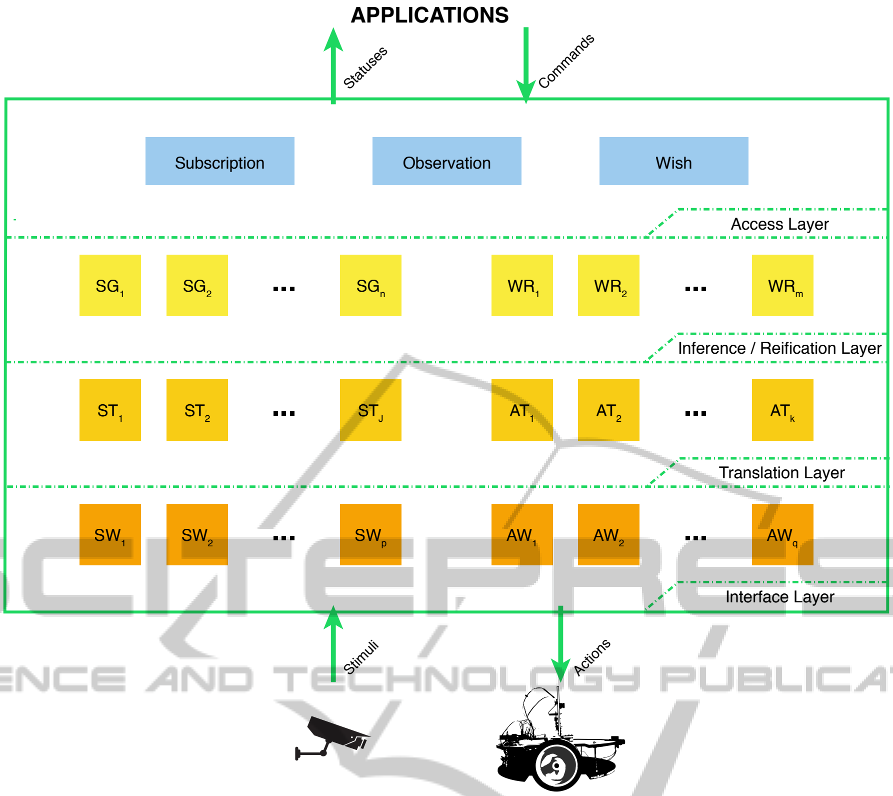

Figure 5: Concrete architecture.

- an *end-user application*, which performs domain specific actions according to the state of the environment model

3.1 DEA Framework

The DEA framework has been developed using Java technologies. In particular, the request endpoints (i.e., the observation, subscription, and wish components) have been integrated with Jess and web services, to deal with the logic predicates and the communication with applications respectively. The communications between the domain-dependant components have been implemented through SIS (Bernini et al., 2012), a publish/subscribe framework based on the multi-space metaphor.

3.2 DEA Platform

The end-user application provides users with an updated information about people location inside a building constituted by two rooms (sal1 and sal2) and a passage connecting them. Moreover, the building is populated by a mobile and controllable entity labeled Poomba (i.e., a robot).

Thus, the environment model consists of the following entities: `Light`, `MobileUnit`, `Person`, and `Room`. Each of them is characterized by the following properties: `Location`, `Type`, and `Id`. Moreover, `Light` has also `OnOffStatus` that states if the light is switched on or off, and `Room` has the `ContainedPeople` property that maintains the number of people that are actually inside the room.

The above entities constitute the whole environment model since they entirely represent the "context of interest". Moreover, the model, by its own nature, handles the entities in the same way and does not specify any structural constraint between them. Possible physical/spatial considerations (e.g., the building topology) have to be done on one or more external physical space models.

The physical environment has been instrumented with RFID sensors produced by Softwork in proximity of the entrance and the exit of each room, with a BTicino system that controls the lights relying on the OWN protocol, and with a mobile unit equipped with actuators (to move the entity) and sensing devices (to perceive stimuli for localization purpose).

The implemented components dealing with domain-related issues are sketched in Figure 6. At a first look they may appear too many, but each

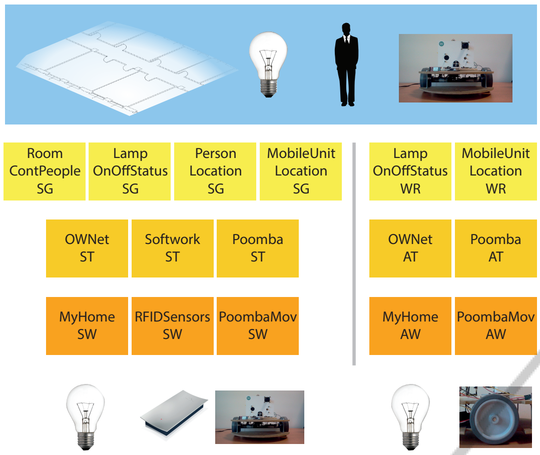

Figure 6: Domain-related components for the applicative scenario.

one is actually very simple, reflecting a philosophy of high-cohesion and low-coupling. In particular, at the interfacing layer, `MyHomeSW`, `RFIDSensorsSW`, and `PoombaMovSW` respectively interfaces with the respective sensors to acquire the generated stimuli; `MyHomeAW` and `PoombaMovAW` respectively interfaces with the respective actuators to deliver actions. At the translation layer, `OWNetST`, `SoftworkST`, and `PoombaST` translate stimuli from the respective sensors into abstract stimuli; `OWNetAT` and `PoombaAT` translate abstract actions from the wish reasoners into actions in a language that the target actuator is able to understand. Finally, at the inference/reification layer, `RoomContPeopleSG`, `LampOnOffStatusSG`, `PersonLocationSG`, and `MobileUnitLocationSG` are in charge of elaborating the abstract stimuli respectively from the `SoftworkST` component to update the `ContainedPeople` property of each `Room` entity, from the `OWNetST` component to update the `OnOffStatus` property of each `Light` entity, from the `SoftworkST` component to update the `Location` property of each `Person` entity, and from the `PoombaST` component to update the `Location` property of the `MobileUnit` entity; `LampOnOffStatusWR` and `MobileUnitLocationWR` are in charge of reifying the commands from the end-user application into corresponding abstract actions and deliver them respectively to `OWNetAT` and `PoombaAT` components.

3.3 End-user Application

The end-user application realizes the following behaviors in response to particular environment conditions:

- If there is at least one person into the sal2 room, all the lights must be switched on
- If there are no people into the sal2 room, all the lights must be turned off
- If there is a professor in one of the two rooms, the mobile unit must be located in the same place

The above behaviors will be realized as follows: the lights will be switched when the first person enters the sal2 room and switched off when the last one exits; in the same way, when a professor enters one of the two rooms, the mobile unit, if not in the same place, will move to reach the room.

The end-user application is a simple Java program that interfaces with the exposed web services of DEA platform. Its basic behavior is to initially send subscription requests and wait for status change notifications to trigger wish requests.

The following, for example, is the subscription request for observing changes in the number of people that are present in the sal2 room:

```
Sender:
   CaseStudyApp
Recipient:
   Type = Room AND
   Name = sal2
SubscriptionRequest:
   ContainedPeople
```

Thus, whenever a new person enters the sal2 room, the application will receive a notification like the following:

```
Sender:
   Id = sal2
StatusChange:
   ContainedPeople 1 0.9
```

This message, for example, notifies that the sal2 room now contains one person with a confidence equals to 0.9. Because there is at least one person in sal2, the application logic triggers a rule that sends a wish request like the following:

```
Recipient:
   Type = Light AND
   Location = sal2
WishRequest:
   OnOffStatus = On
```

This message requests that all the lights in the sal2 room have to be switched on.

It is interesting to notice that the previous notification message could be received both if the sal2 room already contains more than one person or no persons yet. In the first case, the lights are already on and the wish request does not produce any effect; in the second, the lights are switched on. This could be a representative example of delegation of logic.

Finally, to fulfill the last behavior, the application performs the following subscription:

```
Sender:
   CaseStudyApp
Recipient:
   Type = Person AND
   Role = Professor
SubscriptionRequest:
   Location
```

When Mr. White (who is a professor) will enter sal2, the application will receive a message like the following:

```
Sender:
   Id = mr_white
StatusChange:
   Location = sal1 0.9
```

The application then will send the following command:

```
Recipient:
   Id = poomba
WishRequest:
   Location = sal1
```

3.4 Discussion

The design and implementation of the case study application have highlighted the architectural advantages of the solution presented in this paper. In particular, it has pointed out an extreme simplicity in the development.

The necessary knowledge for interacting with the architecture is reduced to sending and receiving well-structured messages, containing data that only refers to the defined entity model and known to the applications. This allows to totally abstract the purely technological and low-level interactions with the devices, enabling the application to interface to an environment representation that is nearer to its logic.

The use of entity models and of an interaction mechanisms linked to logic predications on them also allow the simple adaptation of the application to different environment configuration. The logic the case study application uses is completely unrelated from the concrete environment configuration: for example, the addition of new lights in the sal2 room or the insertion of other professors into the domain entities set does not lead to any modification to the application logic, that remains equally compatible to the requirements. The a-priori assumed knowledge on the domain entities can be easily derived from targeted observation that only exploits the information provided by the model (and possibly by the physical environment structure).

Reasoning on domain data has also made the implementation of the logic less complex, reducing it to an application of simplified rules.

The validation has not highlighted performance issues, an aspect that will be further investigated consequently to a prototype optimization.

4 RELATED WORKS

Devices interoperability is a well known issue (Bonino et al., 2008)(Kusznir and Cook, 2010). Depending on the research field (more or less oriented to hardware integration), the proposed solutions can be classified in two main groups: the former composed by integration platforms that merely unify communication mechanisms from and to devices; and the latter composed by more complex architectures that offer an environment representation to the applications, more suited to the application domain.

DEA fits in the middle of these two classes, reducing their drawbacks and exploiting their advantages.

4.1 Integration Platforms

Solutions in this scope focus on the technological problem of devices interoperability. Typically they offer platforms that abstract specific communication protocols and offer homogeneous mechanisms for interacting with the devices. The general approach is to define a set of communication requirements that applications have to use to interface with them. These requirements are often represented by the use of common vocabularies to uniform syntax and semantics of data and the adherence to a common communication mode.

The *CASAS Lightwight middleware* (CLM) (Kusznir and Cook, 2010), for example, is a solution based on message-passing between information sources (typically sensors) and consumers, that uses the *publish/subscribe* paradigm: the interested components subscribe to specific sources and consequently receive the produced information, expressed by a predefined XML syntax. Thomson et al. in (Thomson et al., 2008) follows a *service-oriented* approach instead, proposing a framework that abstracts devices and exposes them as *web services* or through technologies like Java RMI.

Solutions of this kind often are foundations of research projects, in particular in the field of Ambient Intelligence and Ubiquitous Computing. *CLM*, for example, is used as a base for the communication system of the *smart home CASAS* (that, in the authors knowledge, has not public results yet); the framework by Thomson et al. constitutes instead the device abstraction infrastructure of *Amigo* (Janse et al., 2008), that proposes a *service-oriented* architecture for *smart homes*.

These integration platforms only deal with devices and their data, delegating the end-user applications to fit them into an appropriate environment representation. This allows these solutions to be potentially used in any application domain that concerns hardware components. However, defining and maintaining a proper environment model often is a non-trivial task.

DEA follows the general approach above described and takes inspiration from the CLM publish/-subscribe model for its data abstraction layers, by allowing the communication between device components and inference/reification ones through the SIS framework. The syntax and semantic of the data are defined in a shared vocabulary, that models device raw data in a plain format. In addition, DEA allows to define and maintain an environment model, that could include the architecture into the "domain-oriented architectures" group.

4.2 Domain-oriented Architectures

Domain-oriented architectures mainly focus on offering information models that fit particular application domains. These architectures usually include device interoperability mechanisms, that they use to infer domain knowledge from heterogeneous sources.

Usually information models refer to abstract representation of an environment, whose complexity depends on the specific domain of the solution. In general, solutions in this scope add an abstraction layer to the previous group, with the goal of infer domain knowledge from device data.

In the field of home automation we found, for example, *DOG Gateway* (Bonino et al., 2008), an architecture for "intelligent domotic environments", that abstracts hardware components into an ontological representation of the overall environment, which comprises appliances, various systems (e.g., HVAC, gas, lightning) and simple devices (e.g., lamps), and their spatial location into the environment topology. In this case, the gap between device data and domain knowledge is relatively small: most of the entities at domain level are devices or their aggregations.

In the field of Ambient Intelligence, and in more complex automation solutions, the richness of the models may increase, including more than just device entities and their statuses. In these scenarios, models are enriched by more abstract entities, like people or weather conditions; in general, using an Ubiquitous Computing term, these models deal with *context informations*.

Fernandez-Montes et al. in (Fernandez-Montes et al., 2009) propose a Smart Environments software reference architecture. This architecture implements a *perception-reasoning-action* cyclic flow. Through a component called *Ontologiser*, it organizes data, standardizing them into an environment model. This model includes devices information (i.e., their spatial location and status), inhabitants (like personal data, localization, and health status) and other environment information (e.g., room temperature and brightness). This "perceived" information is used to reason about the environment and then to possibly act on it.

In Ubiquitous Computing, the focus moves further on even more abstract environment representation, where devices may be mere information sources (e.g., RFID tags detecting people presence), thus sometimes directly excluded from the model. An example is *Gaia* (Román et al., 2002), defined by its authors as a middleware for *Active Spaces*. An Active Space is an instrumented environment coordinated by a software infrastructure that extracts context information, that can be useful to adapt the environment itself to the user's needs.

The main common drawback of the solutions of this class is that each of them supports a specific domain and consequently defines a static information model, concretely excluding its reusability in different domains. DEA overcomes this issue by defining a plain and simple method for modeling domain information that is based on property-value pairs. Thus, we define how to model information, but not what, leaving to domain experts or anyone who wants to use the architecture the definition of its own environment model and how device data are linked to it.

Another aspect regards the *action* process, that is, how applications act on the environment to change its state. Although each of the presented architectures model in a clear way the *perception* of the environment (the transformation of device-related data into domain knowledge), there are not details for its symmetric process. The best expectation should be to express actions using the same syntax and semantic of the domain model. DEA complies this expectation by allowing the applications to express wishes on entity statuses, that will be transformed into feasible actions for the appropriate hardware components.

5 CONCLUSIONS AND FUTURE DIRECTIONS

The paper presented an architecture that allows the integration of heterogeneous devices in order to offer end-user applications a representation of the environment at the right level of abstraction. Thus, applications can observe and control the physical environment reasoning only on domain information.

The proposed architecture is independent from the application domain, highly modular and open. In fact it is not designed for a specific scenario, but defines precise levels of abstraction in which placing well-defined components that are domain dependant. Such components are characterized by a high independence and have well-defined interfaces, which specify the structure of the data to be treated and how to communicate with the rest of the architecture. This enforces the openness of the solution, since it encourages the addition of components that adhere to the interfaces and that realize the needed abstraction flows, thus making easy to incrementally support new devices and entity models.

The implementation of a case study has also demonstrated the actual simplification in terms of access to the environment by end-user applications. In particular, the case study has emphasized how the architectural solution allows to clearly separate the implementation strategies (that are responsibility of the applications) from how they perceive and modify the environment (aspects completely managed by the architecture).

Future developments will include the identification of a solution to the problem of the aging of the statuses of the entities. This problem, identified in the analysis stage, regards the updating of the confidence level of the property to which this attribute lapses in the absence of sensory stimuli. Related to this issue, we plan to include the management of histories of the changes. This implies to consider temporal aspects of the information that will be managed using TAM (Time Aware Machine) (Fiamberti et al., 2012), a framework that provides the support in contextualizing information in a temporal context.

REFERENCES

Aiello, M. and Dustdar, S. (2008). Are our homes ready for services? A domotic infrastructure based on the web service stack. *Pervasive and Mobile Computing*, 4(4):506–525.

Bernini, D., Fiamberti, F., Micucci, D., and Tisato, F. (2012). Architectural abstractions for spaces-based communication in smart environments. *Journal of Ambient Intelligence and Smart Environments*, 4(3):253–277.

Bonino, D., Castellina, E., and Corno, F. (2008). The DOG gateway: enabling ontology-based intelligent domotic environments. *Consumer Electronics, IEEE Transactions on*, 54(4):1656 –1664.

Bullivant, L. (2006). *Responsive Environments: Architecture, Art and Design (V&A Contemporaries)*. Victoria & Albert Museum.

Butz, A. and Krüger, A. (2003). A generalized peephole metaphor for augmented reality and instrumented environments. In *Proceedings of The International Workshop on Software Technology for Augmented Reality Systems (STARS)*.

Cook, D. J. and Das, S. K. (2007). How smart are our environments? an updated look at the state of the art. *Pervasive and Mobile Computing*, 3(2):53–73.

Fernandez-Montes, A., Ortega, J., Alvarez, J., and Gonzalez-Abril, L. (2009). Smart environment software reference architecture. In *Fifth International Joint Conference on INC, IMS and IDC, 2009. NCM '09*, pages 397–403. IEEE.

Fiamberti, F., Micucci, D., and Tisato, F. (2012). An object-oriented application framework for the development of real-time systems. *Lecture Notes in Computer Science*, 7304 LNCS:75–90.

Janse, M., Vink, P., and Georgantas, N. (2008). Amigo architecture: Service oriented architecture for intelligent future in-home networks. In *Constructing Ambient Intelligence*, volume 11 of *Communications in Computer and Information Science*, pages 371–378. Springer Berlin Heidelberg.

Kim, J. E., Boulos, G., Yackovich, J., Barth, T., Beckel, C., and Mosse, D. (2012). Seamless integration of heterogeneous devices and access control in smart homes. In *2012 8th International Conference on Intelligent Environments (IE)*, pages 206–213.

Kusznir, J. and Cook, D. (2010). Designing lightweight software architectures for smart environments. In *2010 Sixth International Conference on Intelligent Environments (IE)*, pages 220 –224. IEEE.

Negroponte, N. (1975). *Soft architecture machines*. MIT Press.

Ristau, H. (2008). Publish/process/subscribe: Message based communication for smart environments. In *2008 IET 4th International Conference on Intelligent Environments*, pages 1–7.

Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R., and Nahrstedt, K. (2002). A middleware infrastructure for active spaces. *Pervasive Computing, IEEE*, 1(4):74 –83.

Stevens, W., Myers, G., and Constantine, L. (1979). Classics in software engineering. chapter Structured Design, pages 205–232. Yourdon Press, Upper Saddle River, NJ, USA.

Thomson, G., Sacchetti, D., Bromberg, Y., Parra, J., Georgantas, N., and Issarny, V. (2008). Amigo interoperability framework: Dynamically integrating heterogeneous devices and services. In *Constructing Ambient Intelligence*, volume 11 of *Communications in Computer and Information Science*, pages 421–425. Springer Berlin Heidelberg.