Architectural Issues of Adaptive Pervasive Systems

Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi

Politecnico di Milano
Deep-SE Group - Dipartimento di Elettronica e Informazione
Piazza L. da Vinci, 32 - 20133 Milano, Italy
{caporuscio,funaro,ghezzi}@elet.polimi.it

Abstract. Pervasive systems are often made out of distributed software components that run on different computational units (appliances, sensing and actuating devices, computers). Such components are often developed, maintained, and even operated by different parties. Applications are increasingly built by dynamically discovering and composing such components in a situation-aware manner. By this we mean that applications follow some strategies to self-organize themselves to adapt their behavior depending on the changing situation in which they operate, for example the physical environment. They may also evolve autonomously in response to changing requirements. Software architectures are considered a well-suited abstraction to achieve situational adaptation. In this paper, we review some existing architectural approaches to self-adaptation and propose a high-level meta-model for architectures that supports dynamic adaptation. The meta-model is then instantiated in a specific ambient computing case study, which is used to illustrate its applicability.

Keywords: Pervasive Systems, Software Architecture, Software Evolution, Context-aware Adaptation.

1 Introduction

Many modern advanced applications are developed as pervasive systems that support ubiquitous, continuous and smart interactions among humans, autonomous devices, and the environment, to realize what is often called *ambient intelligence*. Such systems, sometimes also called *open-world systems* [1], are characterized by a highly dynamic software architecture: both the components that are part of the architecture and their interconnections may change dynamically, while applications are running. New components may in fact be created by component providers and made available dynamically. Components may then be discovered, deployed, and composed at run time, removing pre-existing bindings to other components. Applications are often highly distributed, i.e., components are deployed and run on different computational units that may not just be traditional computers, but also appliances, sensing and actuating devices of different kinds. In many cases, the components that constitute an application are

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 492–511, 2010.
© Springer-Verlag Berlin Heidelberg 2010

also operated and run by decentralized and autonomous entities. It has become common to use the terms *services* for such components and *service-oriented architecture (SOA)* to indicate the architectural style. In the SOA case, applications do not have a single ownership and coordination point. Services can only be invoked remotely through their interface. They are otherwise seen from other parts of an application only as black-boxes [13].

Dynamic architectures of the kind we described above are created to support the adaptive and evolutionary situation-aware behaviors that characterize pervasive systems. Sometimes it may be useful to distinguish between *adaptation* and *evolution*. Adaptation refers to the actions taken at run time and affecting the architectural level, to react to the changing environment in which these systems operate. In fact, changes in the physical context may often require the software architecture to also change. As an example, a certain service used by the application may become unaccessible as a new physical environment is entered during execution and a new service may instead become visible. Or a certain service may be changed unexpectedly by the owner of the service and the change may be incompatible with its use from other parts of the application. Evolution instead refers to changes that are the consequence of requirements changes. For example, a 3-D interface becomes available and must be used instead of a previously used traditional interface. In the rest of this paper, most of our examples refer to adaptation, although our approach can also work for certain kinds of evolution. In general, long-lived pervasive systems require that applications should follow some strategies to

- detect the relevant changes in the situation in which they operate, such as the physical environment (or even the changing requirements), and
- react by self-organizing themselves and adapting their behavior in response to such changes.

Adaptive pervasive systems raise many challenges to software engineering. They stress the known methods, techniques, and best practices to their extreme and introduce new difficult problems for which new solutions are needed. The notions of variability and adaptation must permeate all phases, from requirements to design and validation, and even run time. Indeed, the clear and clean traditional separation between *development time* and *run time* becomes blurring. Traditionally, changes are handled off-line, during the maintenance phase. In the new setting, they must be also handled autonomously at run time, as the application is running and providing service. To achieve that, software systems must be able to reason about themselves and their state as they operate, through adequate reflective features available at run time. They must be able to monitor the environment, compare the data they gather against an expected model, and detect possible situational changes. Whenever a deviation is found, an adaptation step must be performed, which modifies the software architecture. For example, the adaptation step might simply perform a new deployment and re-bindings to different components, or re-binding to different external services. In other cases, the adaptation strategies may be more complex.

This paper focuses on software architectures that support run-time adaptation. More specifically, it illustrates a general, high-level reference meta-model and then illustrates how it can be instantiated and adapted to develop a specific case-study in the domain of assisted living.

The paper is structured as follows: Section 2 describes related work. Section 3 illustrates a practical hypothetical example of an adaptive system that serves as a case-study in a pervasive computing setting. Section 4 introduces the most important features of the proposed meta-architecture. Section 5 describes how the concepts of the meta-architecture are instantiated in the proposed solution for the case-study. Finally, Section 6 provides some conclusions and outlines future work directions.

2 Related Work

Research on dynamically adaptable and evolvable software systems became very active in recent years. In the early 2000s, IBM promoted a vision, called *autonomic computing* [10], which focuses on a new generation of software systems that can manage themselves to achieve their goals in a changing world, through *self-configuration*, *self-optimization*, *self-healing*, and *self-protection*. Although an overview of autonomic computing is out of our scope, and would clash with space reasons, we will narrow down the focus of our analysis of related work to architectural solutions enabling self-management. Research in this area has been focusing on two main issues. On the one side, there has been an exploration of the architectural *styles* that best support evolution, due to intrinsic characteristics of the style. On the other, research has focused on the *mechanisms* that can be exploited to achieve adaptation, given a specific architectural model or a specific style. According to [17], "An architectural style defines a vocabulary of components and connector types, and set of constraints on how they can be combined." By focusing on architectural styles, it is possible to focus on evolution from an abstract and high-level viewpoint which may enable systematic and even formal reasoning.

Let us first observe that in the general case, if no specific constraints are assumed on an architecture, a run-time change that requires dynamic updates of components or connectors may require suspension of (parts of) an application to achieve some desirable level of consistency. Managing suspensions can be very complex. This problem has been faced elsewhere in the literature [11,23].

There are styles, however, that facilitate dynamic adaptation. C2 [14] is a well known example of an architectural style that achieves this goal. C2 introduces a sharp distinction between computation and communication and strictly constrains how the application can be built. In particular, every communication among computational units (components) occurs via bus-like connectors, thus minimizing component interdependency. The style also imposes topological constraints: components can consume data from only one connector and produce output data only on one connector. Connectors, instead, can accommodate any number of components or other connectors. Thus every communication is carried

out in an asynchronous way through messages put on and read from connectors. The C2 style is well suited to supporting dynamic adaptation and evolution. The application can be easily modified through the addition and/or removal of components, which can be carried out without suspending any computation.

Other styles than C2 have been scrutinized by Taylor et al. [21] . In that paper, a number of architectural styles used by state-of-art software systems are evaluated according to following criteria:

- How and how much the system's behavior can be changed;
- How long the system's evolution takes to be effective;
- How the state of the system is changed when that system evolves;
- In which environment the system is executing;

Examples of examined styles and corresponding systems are: the publish-subscribe style [7], implemented for example by Siena; the REST style [8] used for web browsing; the CREST style [6] adopted by AJAX and other JavaScript-based technologies.

Regarding the mechanisms that can be superimposed to an architectural model or style to achieve dynamic adaptation, two main research lines emerged so far. The former is about all the approaches that exploit *planning* techniques to cope with unexpected situations, and the second one is based on reactive rule systems. An early example of planning-based approach has been proposed by Traverso and Pistore [22], which synthesizes plans starting from OWL-S process models and a set of prioritized goals. These plans cope with non-deterministic outputs of services by synthesizing `if-then-else` constructs to cover all the possible outputs prescribed by the OWL-S process model [12]. Another planning-based approach is described by Sykes et al. [19,20], where a plan is synthesized starting from goals and available operations. The difference lies in that the non-deterministic output of actions is handled by synthesizing a *reactive* plan. A reactive plan is a plan that for each logical state of the system from which the goal can be reached, prescribes the action to be taken to move towards the goal. In this way, even if the system should deviate from the expected behavior, as long as the goal is reachable by the current state the plan can still suggest a way to achieve the goal.

The other main approach to adaptation is rule-based, e.g., the Rainbow framework [9]. Rainbow is based on the use of a run-time architectural model. Logical probes can be deployed on the running system to gather data useful to enable system evolution and adaptation. Data coming from probes can be aggregated and then used to update the run-time model. To accomplish this task ad-hoc components called *gauges* are introduced. Invariants can be stated about the run-time model and reactive rules can be written to try to enforce them. Reactive rules directly manipulate the run-time architectural model and these changes are automatically reflected in the controlled application.

Another rule-based approach has been developed in our group. It introduces an autonomic element called SelfLet [3] , its model and its developing framework. A SelfLet is characterized by a *goal* and by a course of action (specified through a

finite state machine) to achieve it, called *behavior*. In a world populated by SelfLets, each of them tries to achieve its goal by exploiting functionalities available locally or requesting the needed functionalities to other SelfLets. Such request can be fulfilled both by a SelfLet providing the functionality as a service or by a SelfLet teaching the requestor how to solve the problem. SelfLets are also able to evolve and adapt on the basis of their internal state and the environment they work in. To do so, an *autonomic policy* is specified through reaction rules. Rules can manipulate a SelfLet's behavior by:

- changing the way a service is offered or requested;
- installing a new local service;
- modifying a behavior by adding, deleting or replacing states and transitions in the finite state machine describing the SelfLet.

Combining local *behaviors* and *reactive rules*, a software architect can design a system able to achieve a global goal relying on a SelfLet population making only local decisions. Rule-based approaches are in general computationally lighter than planning-based approaches, but offer only little support to handling unforeseen situations since no reactive rule has been written for them.

An approach, the A3 framework, that cannot be classified neither as planning-based nor as rule-based has also been developed in our group by Baresi and Guinea [2]. A3 is a component-based framework where components are organized in groups, created by application designer to decompose the global task into local sub-tasks. Each group elects a supervisor which both handles communication from and to the group and monitors the group itself. Groups are dynamically created and destroyed and no constraint is imposed on their topology. The grouping dynamism allows to cope with changed situations(e.g., a group with too many components to be effectively handled by a single supervisor). The absence of topological constraints enables the creation of overlapping groups or even hierarchies to more effectively share information among groups. Thus centralized systems' pitfalls are avoided and at the same time the grouping mechanisms make the framework scale very well even for a large number of components.

The approach we illustrate here differs from most of the related work in that it aims at providing a meta-architecture through which the proposed architectural styles and the adaptation mechanisms may be instantiated.

3 Scenario

This section introduces a scenario describing an adaptive embedded system. In particular, it addresses functional evolution of devices embedding on-demand software, namely smart set-top boxes for pay-per-view television. The scenario is further used subsequently to describe the proposed approach.

A *set-top box* is a device that, connected to a source of signal (e.g., telephone line or TV cable) and a television, decodes the signal into contents to be displayed on the television screen. Next generation set-top boxes will also connect to TCP/IP networks by enabling users to browse the web, as well as to access and consume services on-demand.

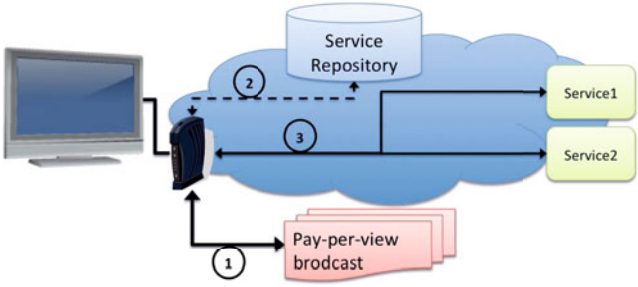

Fig. 1. Scenario

Figure 1 shows a typical scenario where the smart set-top box (1) accesses the contents broadcast by the TV provider, (2) accesses and browses a trusted *Service Repository* (also published by the TV provider) listing the available services, and (3) interacts with the selected services by connecting with the relevant *Service Providers*. When the set-top box is switched on for the first time, it must be configured by the user by selecting the required features. Hence, a basic facility provided by the set-top box supports connection with the TV provider's *Service Repository* (dashed line in Figure 1), a catalog from which the user may select the set of services he or she want to access. As we will see, the *Service Repository* behaves both as a *Registry*, which contains the full descriptions of the services, and as a *Repository* of components that may be downloaded and services provided by the TV provider that can be invoked remotely from the set-top box.

In this setting, possible use cases are: (*i*) *TV on-demand*, where the user chooses which type of contents he or she is interested in, and (*ii*) *Service on-demand*, where the user selects the set of services offered by third-party providers.

3.1 TV On-Demand: Pay-Per-View Home Cinema

The set-top box can access both free and pay-per-view TV contents. While free contents can be directly accessed, in order to consume pay-per-view contents, a user must possess the rights to access the selected channels. In particular, let the user be interested in the "Home Cinema" channel. Then, the actions performed are:

1. User accesses the *Service Repository*, browses the list of available channels and selects the "Home Cinema" pay-per-view channel.
2. Once chosen the channel and paid for it, the set-top box automatically downloads from the *Service Repository* the software component needed to decode the desired contents (e.g., *HC*), which is broadcast in an encrypted format. Indeed, this component is chosen by taking into account both the subscribed channel and the context of the set-top box (e.g., television properties, user requirements).
3. The *HC* component is then deployed into the set-top box.

In this scenario, the set-top box might be reconfigured to support both *software evolution* and *context-aware adaptation*. As an example of evolution, consider a scenario in which a component is downloaded and deployed to improve service fruition (e.g., new codec version, component bug-fix). As an example of adaptation, consider instead the ability of reconfiguring the set-top box with respect to the actual context (e.g., different type of television, server side updates).

3.2 Service On-Demand: eHealth Emergency Management System

As stated above, the set-top box can be used not only to download new contents, but also to access third-party services. For instance, an e-health service might be available to manage health alarms from its subscribers. The user may in this case use the set-top box to send a "health alarm" to the nearest hospital. Let us assume that the service of interest, namely the "eHealth Emergency Management" service (*HEM*), is available in the *Service Repository*. *HEM* is a composite service whose workflow is shown in Figure 2. Namely, it involves a number of standalone activities: (*i*) a *Hospital Yellow Pages* activity (*HYP*) that, given a geographical location, returns a list of the nearest hospitals, (*ii*) the *eHealth Management* (*HM*) provided by the hospital and, (*iii*) a *Display Result* (*DR*) that notifies the workflow results to the final user.

Fig. 2. The "eHealth Emergency Management" workflow

More precisely, the following sequence of steps is performed:

1. User accesses *Service Repository*, browses the list of available services in the entire registry and selects "eHealth Emergency Management" (*HEM*).
2. *HEM* is implemented as a composite service, where the activities mentioned above are provided by means of either local components (e.g., *DR*) or third-party remote services (e.g., *HYP* and *HM*). Hence the *Service Repository* sends to the set-top box the code that implements the workflow of Figure 2 through which both local components and external services are invoked when the user presses a certain button of the remote control.

Also in the case of service on-demand, the set-top box may reconfigure its architecture to achieve both software evolution and context-aware adaptation. As an example of evolution, consider a scenario in which an activity is updated (e.g., a new requirement is added or the activity interaction protocol is modified). On the other hand, as an example of adaptation, consider a scenario in which the set-top box is moved from a location to a different one. By changing the set-top box location (i.e., *context*) the list of the nearest hospitals returned by

HYP changes accordingly and, in turn, the alarm must be sent to a new *HM*. Since the new *HM* could rely on a different interaction protocol – e.g., by means of dial-up – the set-top box needs to be reconfigured in order to *adapt* to the new environment (further discussed in Section 5).

4 A Reference Meta-architecture

In this section we describe the fundamental concepts and properties that characterize software architectures for adaptive pervasive systems. In particular, we crystallize them as a reference high-level meta-architecture that specifies the main distinctive concepts upon which we can define architectural models of self-adaptive systems. Further, Section 5 illustrates a possible architectural solution for the scenario presented in Section 3 and will show how the architectural model conforms to the meta-architecture defined herein. Indeed, the proposed meta-architecture can be instantiated in different ways and even partially in other practical scenarios.

The meta-architecture illustrated in Figure 3 is reminiscent of, and an enrichment of, the abstract structure of an autonomic element [10]. In fact, it defines the main features that characterize a software architecture of applications that may evolve and adapt their functionalities with respect to a change of either their requirements or their surrounding environment.

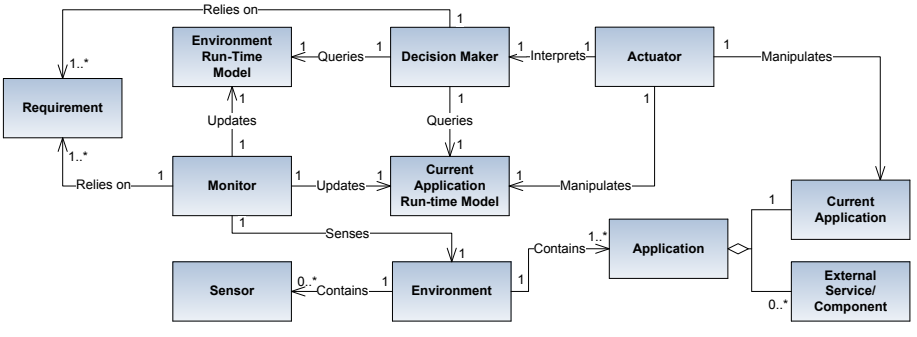

Fig. 3. Meta architecture for self-adaptive systems

The cornerstone of the proposed meta-architecture is the *Requirement* entity, which defines the initial input steering the application assembly, as well as the application run-time behavior. Indeed, it defines the set of properties that an application must satisfy at run time. It is worth to note that in this context, *evolution* refers to the ability of changing requirements at run time, whereas *adaptation* refers to the ability of satisfying the requirements in spite of changes within the execution environment. This twofold role of requirements demands for (*i*) a *Decision Maker* that assembles an abstract description of the application able to satisfy the requirements, and (*ii*) a *Monitor* that is in charge of

collecting data about the application's run-time behavior to verify whether the requirements are satisfied or not during the execution.

The *Decision Maker* is an entity that, relying on *Requirements*, is able to synthesize and assemble an abstract description of the application. The description is abstract in the sense that it does not deal with implementation details but it is a process-like description of the application's behavior specifying (*i*) which activities must be executed to accomplish the task specified by *Requirements*, (*ii*) how the activities interact with one other, and (*iii*) the logic needed to assemble the activities – e.g., referring to the service on-demand scenario in Section 3.2, the *Decision Maker* is in charge of synthesizing the workflow in Figure 2. Furthermore, since these operations must be accomplished also at run time, *Decision Maker* must consider the application's run-time situation to analyze on-the-fly if the application's behavior adheres to the requirements. Specifically, a *situation* is a composite view of both the current state of the application and its surrounding environment. *Decision Maker* retrieves such data by querying *Application Run-time Model* and *Environment Run-time Model*, respectively. If the situation does not satisfy the requirements (e.g., when the set-top box is moved to a new location, the health alarm must be sent to a new hospital), a new abstract description is synthesized and passed to the *Actuator*, which in turn is responsible for assembling and deploying the actual application (*Current Application* – CA). Specifically, *Actuator* interprets the abstract description provided by *Decision Maker* and handles all the technological means needed to build and bootstrap the new application – e.g., locating and accessing the software artifacts implementing the workflow activities in Figure 2. Such separation of concerns makes abstract descriptions technology-agnostic by effectively decoupling the general description of the application from technology-specific actuation. Hence, abstract descriptions generated by *Decision Maker* can be stored and subsequently reused every time they are required, irrespectively of the technology-specific execution environment. As an example, given an abstract description we can generate two equivalent applications, implemented by means of two different technologies, by providing such an abstract description to two different and technology-specific actuators.

Furthermore, to properly make decisions about the run-time reconfigurations, *Decision Maker* needs to query both the application and the environment run-time models. In order to be effective, such models should reflect reality as faithfully as possible. The monitor collects run-time data from the environment; specifically, data gathered from *External Services/Components* that are not part of *Current Application* and sensor data that provide relevant information about the physical environment – e.g., the set-top box position. The result of monitoring can be an update of *Environment Run-time Model* or of *Current Application Run-time Model*. In Figure 4 *Sensor* denotes an abstraction of a device that provides physical context information. Figure 4 also distinguishes between the *Current Application* (CA) and the other *External Services/Components* it interacts with, which may change over time.

5 Scenario Implementation

As introduced in Section 4, the proposed architectural-model for run-time software evolution is centered around the use of a run-time model as an abstraction of the evolving application and the surrounding environment. Further, several entities, making use of such a model, are devised in order to accomplish run-time software evolution. In this section, we apply such a reference meta-architecture to the scenario described in Section 3 and, in particular, we describe how its entities are mapped to this specific use case.

5.1 Application and Environment Run-Time Models

As mentioned earlier, to properly make decisions concerning the dynamic reconfigurations to accomplish, the system must be able to reason about itself and the state of the environment it operates in. To this extent, *Current Application Run-time Model* and *Environment Run-time Model* provide the means through which a reflective behavior may be achieved.

In order to describe a model for the scenario presented in Section 3, we exploit a three layer architectural model (depicted in Figure 4) where (*i*) the *description layer* describes both the application's *functionality* as a workflow containing a sequence of *virtual basic actions* and the *environment* as a set of monitorable *virtual elements*, and (*ii*) the *implementation layer* encapsulates the concrete implementations to which virtual entities may be mapped. Indeed, the *proxy layer* provides a virtualization layer that we use to introduce a further degree of indirection, thus enabling loosely-coupled relations between virtual objects (i.e., basic actions and sensors) and their implementation.

As shown in Figure 4, *description layer* describes both the application model and the environment model. The application model comprises the run-time entities that support the enactment of the behavioral model corresponding to the workflow of the currently active functionality of the set-top box. Since the functionality must be adapted with respect to the actual context, the workflow itself is not directly bound to the concrete implementations of its actions. Rather, dynamic adaptation is achieved by decoupling the virtual basic actions of the workflow at the description level from their concrete implementation, thus achieving the required flexibility supported through dynamic binding.

The *implementation layer* contains the set of all possible components (called implementation components) implementing the virtual basic actions specified within the workflow, as well as other companion components that might be used to support the computation. Each virtual basic action might be implemented by several implementation components that vary from each other in terms of extra-functional properties – e.g., security, performance and reliability. That is, following the Product Line Architecture (PLA) approach [5], the component-based model exploited by both *proxy layer* and *implementation layer* enables components to be specified as "variant", although in our case variant selection is performed at run time. This allows for specifying the alternatives to consider

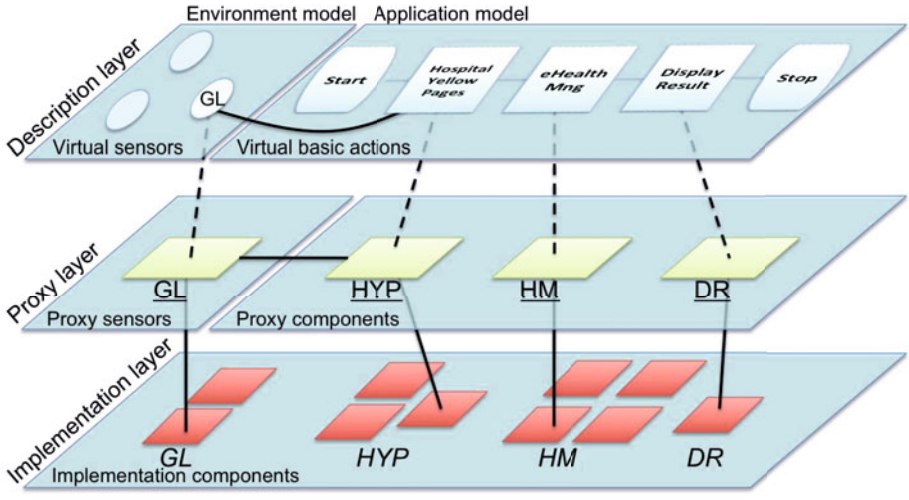

Fig. 4. The three layer run-time model of the application

while mapping virtual basic actions to implementation components. Indeed, alternatives represent the variation points within the run-time model, where the dynamic adaptation of functionality, with respect to the actual needs (i.e., software evolution or context-aware adaptation), can be applied.

In this context, the *proxy layer* plays the role of filtering layer. That is, the components belonging to this layer (called *proxy components*) do not implement the needed functionality themselves. Rather, they implement the logic for variant selection, i.e., they choose among the available components the one that provides the needed functionality and best-fits the requirements – e.g., the most secure, the most efficient, the most reliable. Furthermore, it is worth noticing that no assumption is made about how implementation components are actually implemented. For example, as we will show next, implementation components can be implemented as clients of external and remotely accessible third-party Web Services.

In Figure 4, the *environment model* describes the run-time environment in which the application is executed. In particular, it specifies the set of monitorable *virtual elements* constituting the environment and the data they can provide. The next section describes how such data are made available by means of *proxy sensors* through *implementation sensors*.

5.2 Monitor

Context-aware behaviours and self-reconfiguration require applications to be able to sense the environment and reason about it. Following the meta-architecture presented in Figure 3, the *Monitor* entity is in charge of (i) collecting context data coming from the *Environment* constituents (namely *Sensor* and *Application*) and

(*ii*) updating *Current Application Run-time Model* and *Environment Run-time Model* accordingly.

The *Environment Run-time Model* of Figure 3 maps to *Environment model* of Figure 4, in the *description layer*. This model describes the set of monitorable virtual elements constituting the environment and the data they can provide. As shown in the previous section for virtual basic actions, also virtual sensors are mapped to their corresponding entities in the proxy layer in order to decouple their description from the actual implementations (see Figure 4). That is, each virtual element within the environment is monitored by a proxy sensor belonging to proxy layer.

The workflow specified by the application model in Figure 4, is not environment-agnostic but relies on the set of virtual sensors which constitute the environment model. Indeed, each virtual basic action in the workflow can specify the set of virtual sensors (if needed) to be considered for accomplishing its task. Hence, such a relation must be kept and reflected also at proxy layer where proxy components rely on environmental context data for selecting the specific implementation component to bind with. Hence, each proxy component relates to the proxy sensor needed to gather the environmental context data relevant for selecting the proper implementation component, as well as for computing their tasks.

Furthermore, as done for proxy components, also proxy sensors are implemented as implementation components adhering to the three-layer structure of Figure 4. Also in this case, the separation between proxy sensors and implementation components allows proxy sensors to be implemented by several different implementation components, then allowing them to be dynamically downloaded and deployed when needed.

The next section, describes how such context data, retrieved through *Monitor* and stored within *Environment Run-time Model*, is further queried by *Decision Maker* and used for actuating the application reconfiguration process.

5.3 Decision Maker and Actuator

So far, we have described the entities used by *Decision Maker* to achieve its decisions about run-time reconfigurations, namely those that reify the run-time models. In this section we are going to explain how such decisions are made by *Decision Maker* and further actuated by *Actuator*.

Specifically, referring to the three-layer model described in Section 5.1, decisions that must be made by *Decision Maker* concern the binding (rebinding) between a virtual basic action and the proper implementation component. The proxy component implementing the specific virtual basic action is responsible for this task. The proxy component therefore plays the role of a decision maker. Hence, *Decision Maker* is implemented as the set of *proxy components* deployed at the proxy layer, where each proxy component makes decisions regarding the specific virtual basic action it represents.

While *Decision Maker* retrieves data about the *Environment Run-time Model* through the proxy sensors (see Section 5.2), information about *Application*

Run-time Model are kept by the proxy components themselves and represented as the current binding (i.e., between virtual basic action and implementation component) and the data that have steered the selection of such a binding. That is, every proxy component must know:

- which implementation component has been used during previous executions
- which inputs were received (both by the context and by the previous element in the workflow) that caused that specific binding to be chosen.

The former information is needed to improve performance. In fact, should two identical execution should take place, there would be no binding overhead. The latter information is needed to let proxy components be aware of the fact that a reconfiguration is required. In fact, every change in either the context or the workflow computation will lead to checking if the actual binding is still valid or, otherwise change the binding. Validity of a binding ranges from the mere existence of a target entity that can be reached through the binding, to the fact that the target meets some optimality criteria for quality attributes.

Specifically, a reconfiguration decision might be made to face the following three different cases:

1. A proxy component must select a binding for the corresponding virtual basic action for the first time.
2. A binding between a virtual basic action and its implementation component is no longer valid.
3. The implementation component fails during the execution.

In case no bindings are in place between virtual basic actions and implementation components. The proxy components are in charge of selecting the proper implementation component. The choice is made according to a policy that may take into account extra-functional properties of the available implementation components. In the second case *Decision Maker* must reconfigure the application, to try to still meet the requirements. That is, a new binding must be found in order to accomplish the required task. Finally, in the third case the corresponding proxy component can automatically change the binding using a substitutable implementation component and restarting the computation from scratch for the corresponding virtual basic action. Clearly, if no alternatives are available or all the alternatives fail, the computation specified by the workflow cannot be carried out and an error message is reported to the user.

Following the meta-architecture presented in Section 4, which sharply separates the abstract description of the reconfiguration from its actuation, once the reconfiguration description has been made by *Decision Maker*, it must be passed to the *Actuator*, which in turn will apply it to the current instance of the application. Indeed, in our specific instantiation of the meta-architecture, the proxy component chooses the implementation component that must be used and then passes its reference to the *Actuator* which will perform the following activities:

1. it downloads and deploys the implementation component referenced to the proxy component, if needed.

2. it invokes the implementation component just retrieved by passing the parameters coming from the workflow.
3. it passes back to the workflow the result of the computation coming from the implementation component.

The logic implementing the *Actuator* is provided by the proxy layer itself and is exploited by proxy components every time an invocation must take place.

5.4 OSGi-Based Implementation

Mapping to the proposed reference meta-architecture (see Section 4), in this section we describe how *Application*, *Sensor* and *External Service/Component* entities contained by the *Environment* are actually implemented by means of the OSGi framework.

The scenario presented in Section 3 has been implemented using the OSGi (Open Services Gateway initiative) component-based framework. Many different frameworks have been developed so far for component-based programming such as JavaBeans [18] and COM [16]. Although these systems are widely accepted as standard component-based frameworks, they are not well suited for our purposes, since they do not allow for components addition and removal at run time. More precisely, bindings between components are predefined and fixed, making architectural mutations impossible.

On the contrary, what we need is a framework able to decouple components by achieving a run-time feature that allows both modification of bindings, and components addition and removal. To this extent, the OSGi (Open Services Gateway initiative) [15] is a module system for Java implementing a dynamic component model [4]. At a glance, the core part of OSGi defines (*i*) *bundles* (i.e., components) that can be installed, started, stopped, updated, and removed at run time, (*ii*) the *service registry* that allows bundles to find new services and bind to them, and (*iii*) the *execution environment* that defines methods and classes available within a specific platform (e.g., lower-end device, embedded device, high-end server). As in a service-oriented architecture, an OSGi bundle can publish its services into the *service registry*, making them available to other bundles. The key difference between Web services and OSGi services relies on the fact that while Web services always require some specific transport layer, the OSGi services use direct method invocations. This makes OSGi a well-suited framework for scarce-resource devices.

Referring to the scenario presented in Section 3, Figure 5 depicts a possible implementation of the "eHealth Emergency Management" use case. In particular, the OSGi framework is deployed into the set-top box and contains the set of bundles implementing both proxy components and implementation components that relate to the virtual basic actions specified within the *HEM* workflow (see Figure 2).

Still referring to the meta-architecture presented in Section 4, the *Application Run-time Model* and *Environment Run-time Model* are implemented by means of the three-layer model discussed in Section 5.1, where virtual basic actions

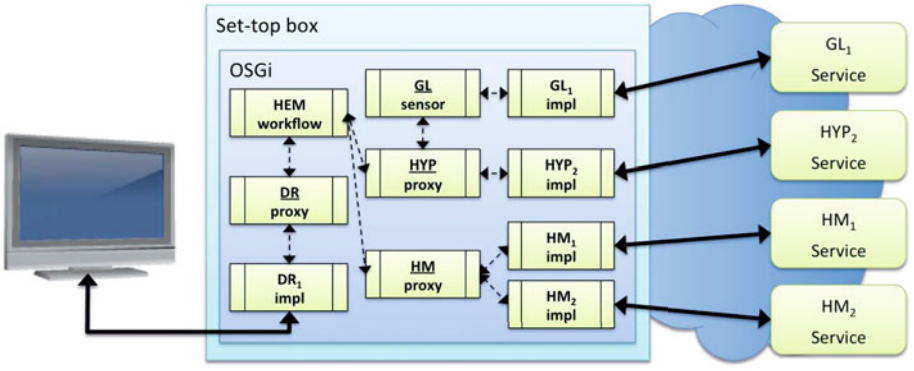

Fig. 5. OSGi implementation of the eHealth Emergency Management System

map straightforwardly to proxy components (i.e., HYP, HM and DR) and their possible implementations, and virtual sensors are monitored by means of proxy sensors. For example, in order to locate the hospital that can better manage the health emergency, the *HYP* virtual basic action needs contextual data about the set-top box geo-location and thus it specifies a dependency with the Geo Location (*GL*) virtual sensor, which, in turn, is implemented as proxy sensor and deployed within the set-top box with the intent of monitoring its position. In particular, referring to Figure 5:

1. A workflow is specified by means of an OSGi bundle (the HEM *workflow bundle*) that invokes the functionalities exposed by the proxy components.
2. A proxy component is implemented by means of an OSGi bundle (the HYP, HM and DR *proxy bundles*), which is responsible for selecting the proper implementation component relying on both extra-functional properties declared by the implementation component and contextual data gathered from proxy sensors.
3. A proxy sensor is implemented by means of an OSGi bundle (the GL *sensor bundle*), which selects the proper implementation component implementing the required monitoring facility.
4. An implementation component is implemented by means of an OSGi bundle (the GL_1, HYP_2, HM_1, HM_2 and DR_1 *impl bundle*), which actually implements the required facility.

In this setting, the HEM workflow is executed as a standard OSGi bundle that invokes the methods exported by HYP, HM and DR proxy bundles, respectively. When invoked, HYP selects the closest hospital by relying on the contextual information provided by GL and returns it to HEM. Clearly GL must be retrieved by the *Service Repository* and installed, unless it is already in place. GL can be in place if another application has already mentioned it as a dependency. After that, GL will search for all the sensors (local or remote) providing the local position and will choose between them according to some policy. To enable HYP to contact GL without the need of hard-coded references, a naming convention

is adopted. That is, proxy sensors have the same name of the corresponding entities in the run-time environment model. In this way the proxy components of every downloaded workflow can easily reference local proxy sensors.

It is important to note that the logic responsible for retrieving contextual data is not application-specific. Rather, once a proxy sensor is downloaded and deployed within the set-top box, it can be accessed and used by many applications at the same time. Once such information has been retrieved, the HYP bundle can select the closest HM hospital and send the alarm to it. The result of such invocation will then displayed on the TV screen through the DR bundle.

As explained above, each *proxy bundle* can select which specific *implementation bundle* to bind to, among the ones that are available. Indeed, this functionality is provided by means of the OSGi late-binding mechanism, which allows for searching, filtering and binding bundles at run time. Specifically, there are three issues regarding such binding selection (refer to Section 5.3): (*i*) a proxy bundle selects the binding to the corresponding implementation bundle for the first time, and (*ii*) the binding between a proxy bundle and its implementation bundle is no longer valid[1]. To face such issues, two strategies are implemented: the former aims at optimizing extra-functional requirements, whereas the second one aims at forcing the application to meet its requirements also in a changed situation.

The first strategy considers extra-functional requirements optimization while binding a virtual basic action to an implementation bundle. For example, when the set-top box must execute the *virtual basic action* corresponding to the *eHealth Manager*, two implementations may be available as possible targets for HM: an *implementation bundle* able to contact the closest hospital's web service or an *implementation bundle* that can contact the hospital via a standard phone call transmitting a pre-recorded vocal message. The two alternatives are functionally equivalent, i.e. delivery of the alarm message to the hospital is guaranteed in both cases. They differ, however, in their extra-functional qualities, such as reliability, performance or cost of connection. A choice can thus be made when the binding is performed according to some optimization strategy. Notice that the same decision making logic can be implemented for the proxy sensors. As an example, the geo-localization proxy sensor could choose between two different implementation bundle, one implemented through a GPS device and the other through an IP-based geo-localization remote service. The two implementation-components may be associated with attributes that specify an accuracy level and a cost; the choice can thus be made by maximizing a quality figure that takes both parameters into account.

On the other hand, the second strategy aims at forcing the application to meet its requirements even in a changed situation. As an example, when the set-top box moved from its original location to a new location, the box's environment changes accordingly. We can also imagine that, during the previous execution of the HEM workflow, the emergency alarm was sent to the closest hospital by

[1] It is worth noticing that, as discussed in Section 5.3, the third issue can be easily solved by reducing it to the first one.

a web service provided by the hospital itself. It may happen that, in the new location, the closest hospital does not provide a web service to handle emergency alarm. It is clear that the emergency alarm must still be delivered, but the way it is delivered must be changed to be adapted to the new environment. Thus the binding established during the previous execution between the eHealth manager virtual basic action and the HM implementation bundle must be changed. Once again, the logic for the adaptation is implemented by a proxy component (HM). In fact, this is the only component in our architecture that knows which is the current binding and which were the inputs from the previous virtual basic action in the workflow, as well as the context that caused such binding establishment. In our use case, given the location change, HM could receive as input a different "closest hospital", and this hospital could support the on-line emergency alarm or not. If the previous communication mode is still supported, then only the web service address must be updated, otherwise a new implementation bundle must be retrieved, downloaded, and deployed to be bound to the eHealth manager virtual basic action.

Notice that a proxy bundle does not directly invoke the selected implementation bundle. Rather, following the meta-architecture an *Actuator* is needed to make the decision application-agnostic. However, in our implementation, the actuator role is played directly by the OSGi framework through the dynamic late-binding mechanism. Indeed, every proxy bundle exploits such a mechanism (that can be logically seen as a proxy layer facility) for selecting and invoking the corresponding implementation bundle, and retrieve the computational results. However, if no implementation bundle matching the requirements is available within the framework, it can be (*i*) downloaded from the TV providers service repository by means of the relative facility provided by the set-top box (see Section 3), (*ii*) published into the OSGi *service registry*, and (*iii*) dynamically invoked at run time.

As few final remarks, concern implementation bundles. They can either implement the required task or be used as a stub to access remote Web Services, which actually implement the tasks. For example, referring to Figure 5, the DR bundle, which depends on the specific TV screen attached to the set-top box, locally implements itself the logic needed to display the workflow outcome on the TV screen. On the other hand, the HM proxy bundle is implemented by a set $\{HM_1, HM_2\}$ of implementation bundles which in turn grant the access to HM_1 Service and HM_2 Service, respectively. This solution allows for reducing the computational burden, which is unbearable for scarce-resource devices such as the set-top box, by delegating the effective implementation to the service provider side.

6 Conclusions

Open-world systems are characterized by a highly dynamic software architecture where both components and their interconnections may change dynamically, while applications are running. Pervasive systems are a notable class of open-world systems, where the need for dynamic software architectures are needed to

support the situation-aware behaviors that characterize them, namely *context-aware adaptation* – run-time actions affecting the architectural level which react to environmental changes – and *software evolution* – changes that originate in the requirements.

In this context, software systems must be able to reason about themselves and their state as they operate, through adequate reflective features available at run time. Moreover, they must be able to monitor the environment, compare the data they gather against the expected model, and detect possible situational changes. Whenever a deviation is found, an adaptation step must be performed, which modifies the software architecture.

This paper presented a meta-architecture supporting run-time adaptation and evolution. In particular, it first described the general, high-level reference meta-model by discussing its constituent entities and the relations between them. Then it illustrated how such a meta-architecture can be instantiated and adapted to develop a specific case-study, namely the "eHealth Emergency Management System". Such a scenario describes an adaptive embedded system – i.e., a smart set-top boxes for pay-per-view television – that addresses the functional evolution and adaptation of on-demand software. We finally demonstrated the applicability of such an approach by implementing the "eHealth Emergency Management System" case study through the OSGi component-based framework, which provides a complete and dynamic component-based programming platform for scarce-resource devices. In particular, we detailed how the entities specified by the abstract meta-architecture have been actually implemented within the concrete "eHealth Emergency Management System" application.

It is worth noticing that the specific application domain addressed herein – i.e., pervasive embedded systems – had an impact on the implementation choices we made, since it presents a set of specific extra-functional requirements. The fact of dealing with resource-scarce devices asks for implementations to be as light as possible, while still satisfying the functional requirements. This is the reason why the abstract application workflow, which might naturally specified by means of a high-level interpreted language (e.g., XML-based), has instead been encoded directly into Java, to reduce run-time overhead. This in fact speeds up execution and eases deployment within the framework by supporting dynamic installation and removal of applications.

Pervasive embedded systems also have to be dependable. An applications like the "eHealth Emergency Management System" must provide an acceptable level service quality even in critical situations. This imposes certain requirements on the *Decision Maker*, which should be able to dynamically reconfigure the bindings to external components and services to achieve the required self-healing capabilities. The proposed solution also ensures a level of security and trust because *implementation components* are provided through a centralized and controlled repository. In a more general case, it might be useful to extend reconfiguration policies beyond just re-binding, e.g. also supporting re-plan of the workflow on-the-fly.

The work described in this paper is part of an on-going long-term research effort that focuses on self-managing situational software systems. Part of this work addresses software architectures that best match the goals of such systems. One possible outcome of this particular work would be the identification of a catalog of architectural solutions that may be adopted in different systems, which would fit the specific characteristics of the systems under consideration. To gain precise, reliable, and reusable knowledge about the various solutions, we are currently engaged in different case studies, ranging from decentralized distributed systems supporting urban mobility scenarios to emergency-management to rescue people in mountain areas. Different architectural solutions will be defined and tried in the case studies, to gain a deep understanding of their potential benefits and drawbacks, and eventually support the development of the solutions catalog mentioned above.

Acknowledgements

This research has been Funded by the European Commission, Programme IDEAS-ERC, Project 227077-SMScom (`http://www.erc-smscom.org`).

References

1. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issue and challenges. Computer 39(10), 36–43 (2006)
2. Baresi, L., Guinea, S.: A-3: Enabling self-adaptation in distributed systems through group abstraction. Technical report, Politecnico di Milano (2009)
3. Bindelli, S., Nitto, E.D., Mirandola, R., Tedesco, R.: Building autonomic components: The selflets approach. In: 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops, ASE Workshops 2008, pp. 17–24 (2008)
4. Cervantes, H., Favre, J.-M.: Comparing javabeans and osgi towards an integration of two complementary component models. In: Proceedings of EUROMICRO Conference (2002)
5. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley Pub. Co., Reading (August 2001)
6. Erenkrantz, J.R., Gorlick, M., Suryanarayana, G., Taylor, R.N.: From representations to computations: the evolution of web architectures. In: ESEC-FSE 2007: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 255–264. ACM, New York (2007)
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
8. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. 2(2), 115–150 (2002)
9. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
10. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer, 41–50 (2003)

11. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng. 16(11), 1293–1306 (1990)
12. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., et al.: OWL-S: Semantic markup for web services (2004)
13. Nitto, E.D., Ghezzi, C., Metzger, A., Papazoglou, M., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Automated Software Engg. 15(3-4), 313–341 (2008)
14. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE 1998: Proceedings of the 20th International Conference on Software Engineering, Washington, DC, USA, pp. 177–186. IEEE Computer Society, Los Alamitos (1998)
15. OSGi Alliance. OSGi service platform, core specification, release 4 (2007)
16. Platt, D.S.: Understanding COM+. Microsoft Press, Redmond (1999)
17. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Englewood Cliffs (April 1996)
18. Sun Microsystems, Inc. JavaBeans, `http://java.sun.com/products/javabeans`
19. Sykes, D., Heaven, W., Magee, J., Kramer, J.: Plan-directed architectural change for autonomous systems. In: SAVCBS 2007: Proceedings of the 2007 Conference on Specification and Verification of Component-Based Systems, pp. 15–21. ACM, New York (2007)
20. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From goals to components: a combined approach to self-management. In: SEAMS 2008: Softw. eng. for adaptive and self-managing systems, pp. 1–8. ACM, New York (2008)
21. Taylor, R.N., Medvidovic, N., Oreizy, P.: Architectural styles for runtime software adaptation. In: WICSA/ECSA 2009 (2009)
22. Traverso, P., Pistore, M.: Automated composition of semantic web services into executable processes. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 380–394. Springer, Heidelberg (2004)
23. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans. Softw. Eng. 33(12), 856–868 (2007)