Architectural Patterns for Context-Aware Services Platforms

P. Dockhorn Costa, L. Ferreira Pires and M. van Sinderen

Centre for Telematics and Information Technology, University of Twente,
PO Box 217, 7500 AE Enschede, the Netherlands

Abstract. Architectural patterns have been proposed in many domains as means of capturing recurring design problems that arise in specific design situations. In this paper, we present three architectural patterns that can be applied beneficially in the development of context-aware services platforms. These patterns present solutions for recurring problems associated with managing context information and proactively reacting upon context changes. We demonstrate the benefits of applying these patterns by discussing the AWARENESS architecture.

1 Introduction

Architectural patterns have been proposed in many domains as means of capturing recurring design problems that arise in specific design situations. They document existing, well-proven design experience, allowing reuse of knowledge gained by experienced practitioners [1]. For example, a software architecture pattern describes a particular recurring design problem and presents a generic scheme for its solutions. The solution scheme contains components, their responsibilities and relationships.

Patterns for software architectures also exhibit other desirable properties [1]: (i) patterns provide a common vocabulary and understanding for design principles; (ii) they are a means for documenting software architectures; (iii) they support the construction of software with defined properties; (iv) they support building complex and heterogeneous software architectures; and (v) they help managing software complexity.

In this paper, we present three architectural patterns that can be applied beneficially in the development of context-aware services platforms, namely the Event-Control-Action pattern, the Context Sources and Managers Hierarchy pattern and the Actions pattern. These patterns present solutions for recurring problems associated with managing context information and proactively reacting upon context changes.

A context-aware services platform [5] contains generic components to support development, deployment and execution of context-aware applications. Examples of functionality provided by such components include context management (gathering and processing context information), reactivity upon context changes, and 3rd party service usage. The reuse of services is the most emphasized benefit of using services platforms support. Embedding common complex tasks into the platform and making

Dockhorn Costa P., Ferreira Pires L. and van Sinderen M. (2005).
Architectural Patterns for Context-Aware Services Platforms.
In *Proceedings of the 2nd International Workshop on Ubiquitous Computing*, pages 3-18
DOI: 10.5220/0002573800030018

them uniformly available for reuse can greatly enhance the efficiency of application development.

The approach chosen to present these patterns has been inspired by the book [8], which describes a pattern as a three-part scheme: (i) a situation giving rise to a problem; (ii) the recurring problem arising in that situation and (iii) a proven solution to the problem. Therefore, for every pattern presented in this paper, we discuss (i) an example situation where the problem occurs; (ii) the recurring problem being considered; (iii) the solution scheme for this problem containing structural aspects with components and relationships and dynamic (behavioral) aspects and (iv) the general benefits of applying this pattern.

The remainder of this paper is structured as follows: Section 2 presents the Event-Control-Action Pattern, Section 3 discusses the Context Sources and Managers Hierarchy pattern and Section 4 describes the Actions pattern. Section 5 discusses the applications of these patterns in a (partially) prototyped services architecture, Section 6 presents related work and Section 7 gives final remarks and identifies topics for further study.

2 Event-Control-Action Pattern

The Event-Control-Action architectural pattern provides a high level structure for systems that proactively react upon context changes. It has been devised in order to decouple context concerns from reaction (communication and service usage) concerns, under control of an application model. An application model defines the behavior of the application, which may be described by means of, for example, condition rules. In this pattern, context management issues, such as sensing and processing context, are independent from issues regarding reacting upon context changes.

2.1 Example

Suppose our services platform needs to provide support for applications in the medical domain. An example of such an application would be a tele-monitoring application [3], which monitors epileptic patients and provides medical assistance moments before and during an epileptic seizure. Measuring heart rate variability and physical activity, this application can predict future seizures and contact relatives or healthcare professionals automatically. In addition, the patient can be informed moments in advance about the seizure, being able to stop ongoing activities, such as driving a car or holding a knife. The aim of using this system is to provide the patient with both higher levels of safety and independence allowing him to function more normally in society despite his disorder.

In this system scenario, the patient wears a heart monitoring system that collects heart signals along the day. These signals are processed by smart algorithms which are able to detect abnormalities, such as the possibility of having an epileptic seizure, within seconds.

Several actions may be taken upon an epileptic seizure: (i) a volunteer, normally an intimate of the patient capable of providing first aid, receives an alarm of a possible seizure, (ii) in case no volunteer is available, healthcare professionals are sent to his location, (iii) patient's bio-signals derived from the monitoring system are streamed to doctors at real time, and (iv) based on the real time information, doctors decide whether the patient needs to be taken to the nearest hospital.

2.2 Problem

The example presented in section 2.1 imposes challenging requirements to the support platform:

- The platform should offer support for gathering context information, such as the patient's heart rate and blood pressure in order to predict possible epileptic seizures;
- The patient's and volunteers' locations need to be known, and proximity information needs to be derived;
- Full time connectivity with the patient needs to be provided;
- Devices (e.g., mobile phones) of volunteers and doctors need to pass an alarm in case of seizure;
- Real time streaming connections need to be established with the doctor;
- In case of a critical situation, an ambulance needs to take the patient to the nearest hospital.

Implementing such an application within a single business party is not feasible. In fact, this application is realized with the cooperation of several business parties: the location providers, the providers of algorithms to analyze heart rates, the doctors clinic, the hospital, the connectivity providers, and the manufacturers of monitoring devices, among others. The aim of the platform is to guarantee the execution of the application by configuring and coordinating the cooperation of functions distributed among business parties. The distribution of responsibilities among these parties and the coordination of distributed functions require agreements on certain architectural patterns.

2.3 Solution

The Event-Control-Action architectural pattern aims at providing a structural scheme to enable the coordination, configuration and cooperation of distributed functionality within services platforms. It divides the tasks of gathering and processing context information from tasks of triggering action in response to context changes, under the control of an application behavior description. We assume context-aware application behaviors are described in terms of condition rules, such as `if <condition> then <actions>`. The condition part specifies the situation under which the actions are enabled. Conditions are represented by logical combinations of *events*. An event models some *happening of interest* in our application or its environment. The observation of events is followed by the triggering of actions, under control of

condition rules. Events are modeled and observed by one or more Context Processor components.

A Controller component, empowered with condition rules describing application behaviors, observes the events. In case the condition turns `true`, an Action Performer component triggers the actions specified in the condition rules. Actions are operations that affect the application behavior in response to the situation defined in the condition part of the rule. An action can be a simple web services call or a SMS delivery, or it can be a complex composition of services.

2.4 Structure

Fig. 1 shows a class diagram of the Event-Control-Action pattern as it is supposed to be applied in a context-aware services platform.

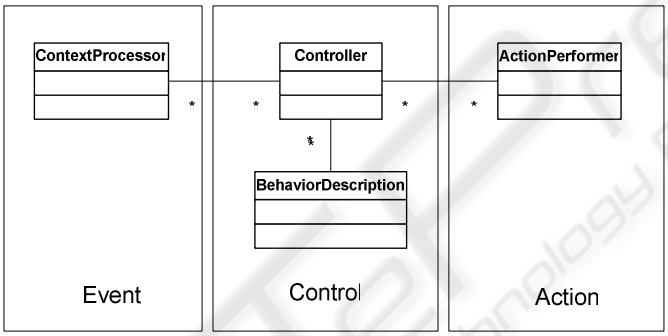

Fig. 1. Event-Control-Action pattern.

Context concerns are placed on the left side of the figure, which depicts the Context Processor component. This component depends on the definition and modeling of context information. The Controller component, positioned in the central part of the figure, is provided with application behavior descriptions, represented by the Behavior Description class. On the right side of the figure, the action concerns are addressed. The Action Performer component triggers actions, which can be a service invocation on (external or internal) service providers or a network.

2.5 Dynamics

Consider the example presented in Section 2.2 in which a possible epileptic seizure is detected and volunteers close to the patient are contacted via SMS. We assume in this scenario that the services platform has been correctly configured (condition rules are defined, devices are switched on and users are subscribed to required services).

Fig. 2 depicts the flow of information between components applying the Event-Control-Action pattern.

The condition rule (here applied to a patient called John) defined within the Controller is:

```
if <event:EpilepticAlarm>
then
     <SendSMS(closeby(Volunteers, 100))>
```

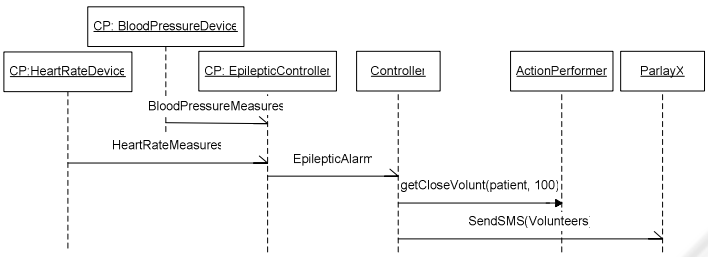

Fig. 2. Dynamics of the event-control-action pattern.

The Controller observes the occurrence of event *EpilepticAlarm*. This event is captured by the component Epileptic Controller, which is an instance of Context Processor. Blood pressure and heart rate measures are gathered from other dedicated instances of Context Processor. Based on these measures and a complex algorithm, the Epileptic Controller component is able to predict within seconds that an epileptic seizure is about to happen, and an EpilepticAlarm event is, therefore, generated.

Upon the occurrence of event *EpilepticAlarm*, the Controller triggers the action specified in the condition rule. The action `SendSMS(closeby(volunteers, 100))` is a composed action that can be partially resolved and executed by the platform. The inner action `closeby (volunteers, 100)` may be completely executed within the platform. However, it is the responsibility of the services platform designers to decide whether this is meaningful and worthwhile. The execution of this action requires another cycle of context information gathering on Context Processors (to provide the current location of John and his volunteers and to calculate proximity of these persons). By invoking the operation `getCloseVolunt(John, 100)` with assistance of an internal Action Performer, the Controller is able to obtain the volunteers that are within a radius of 100 meters from patient John. Finally, the Controller remotely invokes an action provided by a third party business provider (e.g., a Parlay X [9] provider) to send volunteers SMS alarm messages.

2.6 Benefits

By applying the classic design principle of separation of concerns, the Event-Control-Action pattern has effectively enabled the distribution of responsibilities in context-aware services platforms. Context Processor components encapsulate context related concerns, allowing them to be implemented and maintained by different business

parties. Actions are decoupled from control and context concerns, permitting them to be developed and operated either within or outside the services platform.

Applying such design principles greatly improves the extensibility and flexibility of the platform, since context processors and action components can be developed and deployed on demand. In addition, the definition of application behavior by means of condition rules allows the dynamic deployment of context-aware applications and permits the configuration of the platform at run-time.

3 Context Sources and Managers Hierarchy Pattern

The Context Sources and Managers Hierarchy architectural pattern provides a hierarchical structure for Context Processor components. It has been devised in order to recursively apply context information processing operations in a hierarchy of Context Processors. In this chain of context information processing, the outcome of a context processing unit becomes input for a higher level unit in the hierarchy until a top-level unit is reached.

3.1 Example

Suppose we are developing the same system scenario presented in Section 2.2, in which a possible epileptic seizure is predicted. In addition to contacting nearby volunteers, we would like to know whether the patient is driving in order to send him a personalized alarm, such as “please, stop the car as soon as possible, your may have an epileptic seizure”.

3.2 Problem

Processing context information is challenging. Deducing rich information (e.g., an epileptic alarm) from basic sensor samples (e.g., heart rate and blood pressure measures) may require complex computation. There may be several information processing phases needed before yielding (syntactically and semantically) meaningful context information.

Context information processing activities include [4]:

- Sensing: gathering context information from sensor devices. For example, gathering location information (latitude and longitude) from a GPS device;
- Aggregating (or fusion): Observing, collecting and composing context information from various context information processing units. For example, collecting location information from various GPS devices;
- Inferring: interpretation of context information in order to derive another type of context information. Interpretation may be performed based on, for example, logic rules, knowledge bases, and model-based techniques. Inference occurs, for instance, when deriving proximity information from information on multiple locations;

- Predicting: the projection of probable context information of given situations, hence yielding contextual information with a certain degree of uncertainty. We may be able to predict in time the user's location by observing previous movements, trajectory, current location, speed and direction of next movements.

The services platform should provide mechanisms to distribute context processing activities among multiple components. In addition, it should be able to create compound context information based on various context information sources. Distribution and composition of context information components in a flexible and decoupled fashion require agreements on architectural decisions.

3.3 Solution

The Context Sources and Managers Hierarchy architectural pattern aims at providing a structural schema to enable the distribution and composition of context information processing components. We define two types of Context Processor components, namely Context Sources and Context Managers. Context Sources components encapsulate single domain sensors, such as a blood pressure measuring device or a GPS. Context Manager components cover multiple domain context sources, such as the integration of a blood pressure and heart rate measures. Both perform context information processing activities as mentioned in Section 3.2.

The structural schema proposed by this pattern consists of hierarchical chains of Context sources and Managers, in which the outcome of a context information processing unit may become input for the higher level unit in the hierarchy. The result structure is a *directed acyclic graph*, in which the initial vertexes (nodes) of the graph are always Context Sources components and end vertexes may be either Context Sources or Context Managers. The directed edges of the graph represent the (context) information flow between the components. We assume that cooperating Context Source and Manager developers have agreed upon the semantics of information.

3.4 Structure

Fig. 3 (left side) zooms in the Event part of Fig. 1. It shows a class diagram of the Context Source and Manager Hierarchy pattern as it is supposed to be applied for context-aware services platforms.

Context Managers inherit the features of Context Sources, and implement additional functions to handle gathering context information from various Context Sources and Managers. A Context Managers observes context from one or more Context Sources and possibly other Context Managers. The association between the Context Manager class and itself is irreflexive. Fig. 3 (right side) depicts a directed acyclic graph structure, which is an instantiation of this pattern. CS boxes represent instances of Context Sources and CM boxes represent instances of Context Managers.

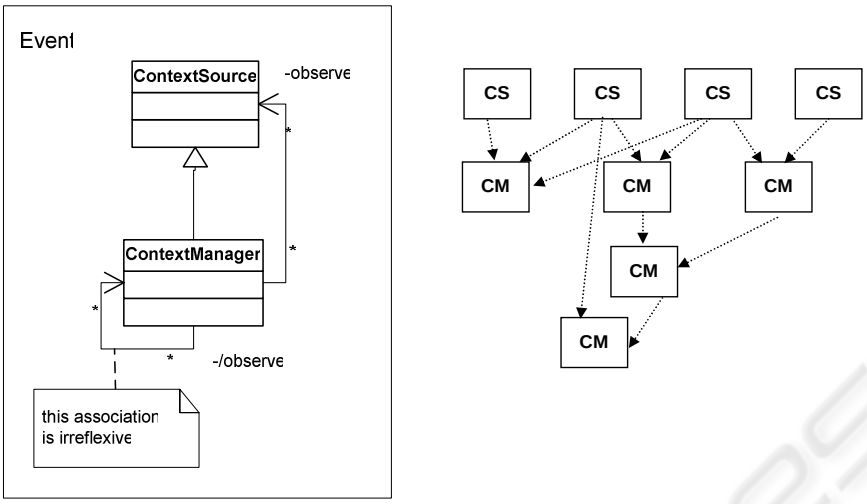

Fig. 3. Context Sources and Managers Hierarchy pattern on the left and an instantiation of this pattern on the right.

Within a single context information processing unit (Context Source or Manager), we verify recursive applications of the Event-Control-Action pattern (Section 2). Consider the following application condition rule manipulated by Controller C1 in Fig. 4:

```
if <event:(EpilepticAlarm ^ driving)>
then
   <SendSMS("please, stop the car as soon as possible, your
may have an epileptic seizure")>
```

The event (EpilepticAlarm ^ driving) is a compound event observed on the following components: (i) a Context Manager component that detects an epileptic alarm (E1 in Fig. 4) and (ii) a Context Source component that detects the patient is driving[1] (E2 in Fig. 4). Within the epileptic detector Context Manager (E1), the following condition rule[2] is described in Controller C2, characterizing the recursive nature of the Event-Control-Action pattern:

```
if <event:(HeartRate > threshold)>
then
   <generate(EpilepticAlarm)>
```

Controller C2 observes heart rate measures on a Context Source component. The action of this rule is the generation of the epileptic alarm signal. Within the driving detector Context Source (E2), the following condition rule is described in Controller C3:

[1] We assume there is a sensor in the patient's car to detect whether he is driving.

[2] For the sake of the example, we simplify the algorithm to detect epileptic seizures by specifying it as the verification of heart rate measures against a threshold value. This algorithm in reality is surely more complex than the simple value comparison given in this condition rule.

```
if <event:(userSignalOn)>
then
   <generate(driving)>
```

The event userSignalOn may be directly set by the patient or automatically sensed by a device embedded in the car that is able to detect his presence.

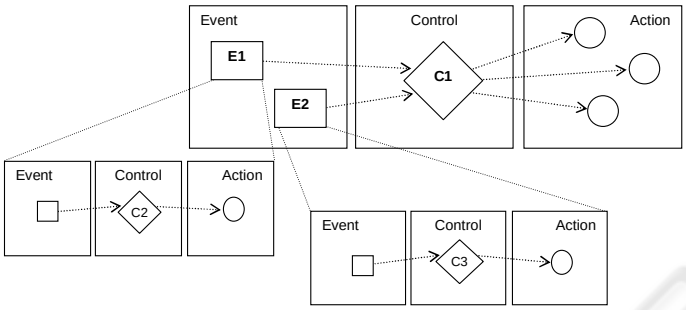

Fig. 4. Recursive application of the Event-Control-Pattern.

3.5 Dynamics

Consider the example discussed in the previous sections. We assume the services platform has been correctly configured (condition rules are defined, devices are switched on, users are subscribed to required services and Context Sources and Managers structures have been configured).

Fig. 5 (left side) depicts the flow of information between components in the Context Sources and Managers structure at the top most application of the Event-Control-Action pattern. At this level, ControllerC1 observes the occurrence of event `(EpilepticAlarm ^ driving)`, which is generated from CM: `EpilepticDetector` and CS: `DrivingDetector`, respectively. When the condition turns true (the alarm has been launched and the patient is driving), the personalized SMS message is sent to the patient.

In the second recursion level of the event-control-action pattern (Fig. 5, right side), the ControllerC2 observes heart rate measures from a heart device Context Source. Empowered with algorithms able to detect heart rate abnormality, the controller generates the `EpilepticAlarm` when it detects the possibility of an epileptic seizure.

3.6 Benefits

The Context Sources and Managers architectural pattern defines a hierarchical structure reference for Context Source and Manager components. This approach has enabled encapsulation and a more effective, flexible and decoupled distribution of context processing activities (sensing, aggregating, inferring and predicting). This attempt improves collaboration among context information owners and it is an

appealing invitation for new parties to join this collaborative network, since collaboration among more partners enables availability of potentially richer context information.

Another important benefit of applying this pattern is that it enables filtering of unnecessary information across the hierarchy of context information processing units. At the lowest level of context information gathering, a great overhead of information flow can be detected but only the relevant information is kept and forward to the next level of the hierarchy.

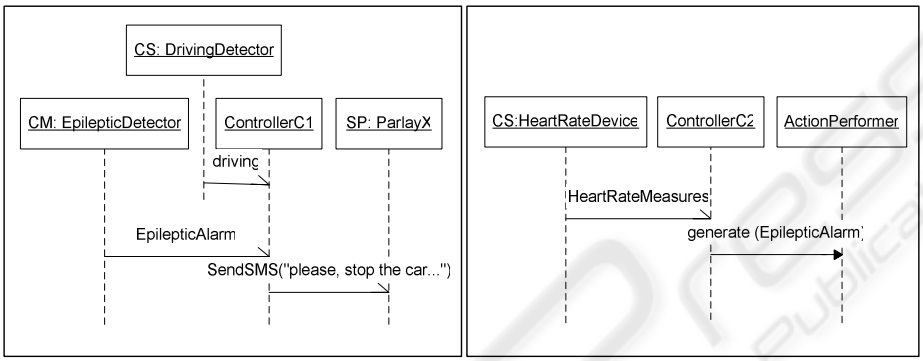

Fig. 5. Dynamics of the Context Sources and Managers pattern on the highest level of the event-control-action pattern recursion (on the left), and on the second level of recursion (on the right).

4 Actions Pattern

The Actions architectural pattern provides a structure of components to support designing and implementing action concerns within context-aware services platforms. It has been devised in order to decouple action purposes from action implementations and to coordinate composition of actions. An action purpose defines an abstract action intention, while its implementation represents the realization of this intention utilizing specific implementation technologies.

4.1 Example

Consider the scenario presented in Section 2.2 in which the actions taken upon an epileptic seizure alarm are (i) a warning message is sent to the patient; (ii) his close relatives are called, (iii) volunteers close to the patient are notified of a possible seizure, and (iv) in case no volunteer is available, healthcare professionals are sent to the patient's current location.

4.2 Problem

Some of the actions presented in the example may be performed independently in parallel, such as (i) sending a warning message to the patient, (ii) calling the relatives and (iii) notifying nearby volunteers. However, the action to call healthcare professionals is only enabled in case notifying nearby volunteers (action (iii)) has not succeeded (for example, no volunteers are momentarily available). This situation characterizes a dependency between actions. In addition, some actions may trigger a sequence of other actions. For instance, to send help from healthcare professionals, it may be necessary to request the patient's medical dossier, to select relevant medication, to check availability of transportation, and so forth.

The services platform should provide mechanisms to manage coordination of actions, especially when dependencies exist. In addition, the platform should support decoupling of an action purpose from its implementations. Although the action "send healthcare professionals" presents a common purpose, the implementations of it may vary, since the logistics may differ from hospital to hospital. Distribution and coordination of actions in a flexible and decoupled fashion require agreements on architectural decisions

4.3 Solution

The Actions architectural pattern aims at providing a structural scheme to enable coordination of actions and decoupling of action implementations from action purposes. It involves (i) an Action Resolver component that performs coordination of dependent actions, (ii) an Action Provider component that defines action purposes and (iii) an Action Implementor component that defines action implementations.

An action purpose describes an intention to perform a computation with no indications on how and by whom these computations are implemented. Examples of action purposes are "call relatives" or "send a message". The Action Implementor component defines various ways of implementing a given action purpose. For example, the action "call relatives" may have various implementations, each defined by a telecom provider. And finally, the Action Resolver component applies techniques to resolve compound actions, which are decomposed into indivisible units of action purposes (indivisible from the platform standpoint).

4.4 Structure

Fig. 6 zooms in on the Action part of Figure 1. It shows a class diagram of the Actions pattern as it is supposed to be applied for context-aware services platforms.

Both the Action Resolver and Action Provider components inherit the characteristics of the Action Performer component, and therefore are both enabled to perform actions. The Action Resolver component performs compound actions, decomposing them into indivisible action purposes, which are further performed separately by the Action Provider component. Action Providers may be

communication service providers or (application) service providers. Communication service providers perform communication services, such as a network request, while service providers perform general application-oriented services, implemented either internal or external to the platform, such as an epileptic alarm generation or an SMS delivery, respectively.

An action provider may aggregate various Action Implementor components, which provide concrete implementations for a given action purpose (represented by implementors A and B in Fig. 6).

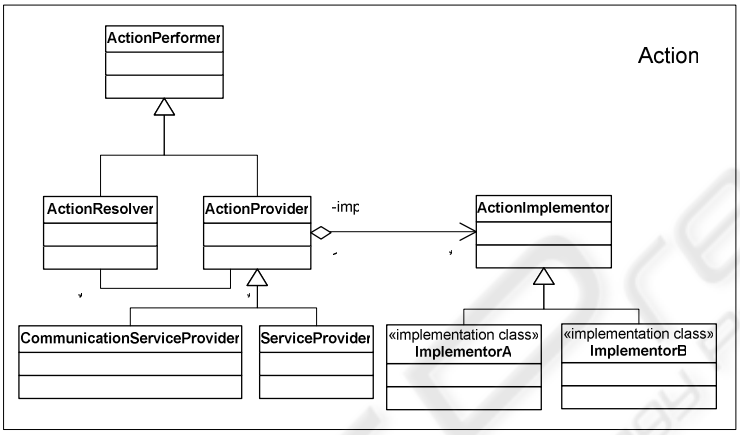

Fig. 6. Actions pattern structure

4.5 Dynamics

Fig. 7 depicts the flow of information between components of the Actions pattern for the scenario presented in Section 4.1.

The Action resolver gets a compound action to decompose. Empowered with techniques to solve composition of services, the Action Resolver breaks the compound action into indivisible service units, which are then forwarded to the Action Provider. The Action Provider delegates them to the proper concrete action implementations. In our example, send SMS and calling actions are delegated to the ParlayX implementor and the action to send healthcare is delegated to the hospital implementor.

4.6 Benefits

By defining a structure of Action Resolvers, Providers and Implementors, the Actions pattern has enabled the coordination of compound actions and the separation of abstract action purpose from its implementations. This attempt avoids permanent binding between an action purpose and its implementations, allowing the selection of different implementations at platform run-time. In addition, abstract action purposes

and concrete action implementations may be changed and extended independently, improving dynamic configuration and extensibility of the services platform.

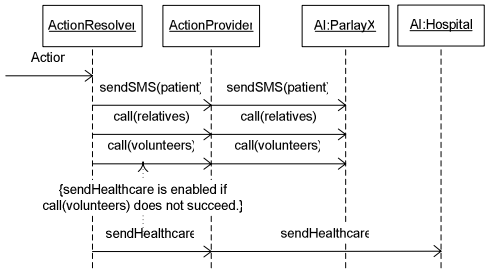

Fig. 7. Dynamics of Actions Pattern

5 The AWARENESS Architecture

The AWARENESS project [7] aims at researching and designing a services and network infrastructure for context-aware applications. The general architecture consists of three layers: the application layer, the services infrastructure layer and the network infrastructure layer.

The application layer defines applications to be developed, deployed and executed using the support from the services and network infrastructure. In AWARENESS, the application scenario chosen to validate the infrastructures is the mobile health application that supports monitoring of epileptic seizures and uncontrolled movements in spasticity [3].

The network infrastructure provides context-aware mobility support in dynamic network environments. Context information, such as presence and available bandwidth, is used to provide dynamic network routing and network selection. The services infrastructure provides generic support easy and rapid development, deployment and execution of context-aware applications. The functionality provided by the service infrastructure include context management (gathering, processing and reacting upon context changes) federated identity management, 3rd party service usage, service discovery, privacy enforcement and security mechanisms.

Although the network layer also benefits from the patterns presented in this paper, we have focused on their applicability in the services infrastructure layer. Fig. 8 presents the AWARENESS services infrastructure architecture.

The *Context Sources* and *Managers* components address context specific concerns such as gathering, processing, and delivering context information. The *Control* module contains the application-specific functionality that is executed within the service infrastructure. The *Actions* module concentrates the functionality to trigger actions in response to context changes. The commands to trigger action are activated by the Controller component.

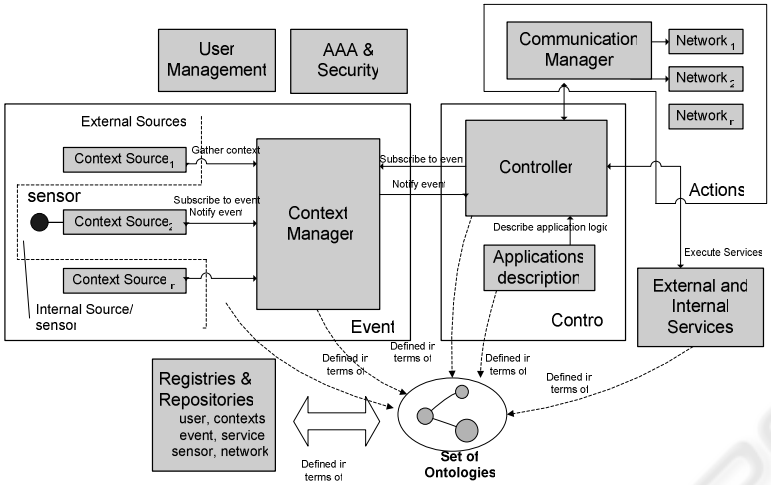

Fig. 8. The AWARENESS services infrastructure architecture

The *Registries & Repositories* module contains information on context types, event types, services and other (meta) information on types and services within the service infrastructure. It interacts with most of the components within the infrastructure since it defines the data types manipulated by them. The *User Management* module contains functionality related to access control, privacy and group management. User profiling and preferences functions are managed by this component.

The *AAA & Security* module includes the functions associated with security, privacy, authentication, authorization (including 3rd party access control), accounting and federation issues. Although this module is depicted in a single block, AAA & Security are cross-cutting issues, meaning that these concerns influence the development of other components of the infrastructure.

The application of the Event-Control-Model architectural pattern is depicted in the figure. Context concerns (Event module) are decoupled from communication and service usage concerns (Action module), under control of an application model (Control module). Applications describe applications logic to the Controller (Control module). Applications logic descriptions specify conditions under which *actions* (services) are to be triggered. The conditions are specified in terms of (correlation of) *events*. Events are modeled and observed by Context source and Manager components. When the conditions hold (events have been observed), the controller triggers the specified actions.

Context Source and Manager components are hierarchically organized, characterizing the application of the Context Sources and Managers Hierarchy pattern. Context Source components encapsulate single domain sensors, while a Context Manager component covers multiple domain context sources. Context Source and Manager components may be implemented as part of the infrastructure or externally provided by third party context providers. Both types of components may implement aggregation, prediction and inferring functionality. Aggregation, prediction and Inferring rules can be described in terms of ontology languages [6].

We are currently applying, implementing and validating the three presented patterns in the tele-monitoring scenario [3].

6 Related Work

The Event-Control-Action pattern inherits properties of design patterns presented in the literature, such as Observer and Mediator [2]. The Observer design pattern defines "dependencies between objects so that when one object changes state, all its dependents are notified and updated automatically". In the event-control-action pattern, the control concerns can be seen as an observer and the event concerns, the subject of observation. The publish-subscribe mechanism can be used to implement observation: the subject is the publisher of notifications and the observers are the subscribers to receive notifications. Any number of observers can subscribe to receive notifications and the subject does not need to know its observers.

The Mediator design pattern allows loose coupling interactions by keeping objects from referring to each other explicitly. It can be used in combination with the Observer pattern to decouple subject from observers: rather than subscribing and notifying directly to objects, subscriptions and notifications are submitted to an intermediate object (e.g., an event channel). Finally, the Actions pattern extends the Bridge design pattern [1], which "decouples an abstraction from its implementations so that the two can vary independently".

In [5] we report the design of a configurable context-aware services platform and in [6] we discuss the utilization of ontology techniques in context-aware systems.

7 Conclusions

We have presented in this paper three architectural patterns that can be beneficially applied in the development of context-aware services platform, namely the Event-Control-Action pattern, the Context Sources and Managers Hierarchy pattern and the Actions pattern. By decoupling context concerns from action concerns, the Event-Control-Action pattern has effectively enabled the distribution of responsibilities among various business parties within a context-aware services platform. This approach has greatly improved extensibility and flexibility of the platform's generic functionality. The Context Sources and Managers Hierarchy pattern enables a flexible and dynamic distribution of context information processing activities within a collaborative network of context sources and managers. Finally, the Actions pattern defines a structure of action performer components that enables the coordination of compound actions and the separation of abstract action purposes from their implementations. This attempt allows the dynamic selection of action implementations at run-time, improving extensibility and flexibility of the platform (3rd party services can be developed and deployed on demand at platform run-time).

Additionally to the benefits just mentioned, the application of these patterns in the AWARENESS project has improved the collaboration among project members. By defining and applying patterns we have provided a common vocabulary and

understanding of concerns within the project. Therefore, project documents produced by different partners have become more consistent. In general, the application of patterns has helped us managing the heterogeneity and complexity of the AWARENESS services infrastructure, since we were able to compose an architecture of distributed collaborative components, which can be developed and maintained by various business parties.

Further study topics should include (i) the definition of an expressive language to define condition rules; (ii) the specification of a mechanism to allow automatic configuration of condition rules into the context sources and managers hierarchy; and the (iii) definition of concrete techniques to resolve composition of actions

Acknowledgements

This work is part of the Freeband AWARENESS project [7]. Freeband is sponsored by the Dutch government under contract BSIK 03025.

References

1. Buschmann, F., et al.: Pattern-Oriented software architecture: A System of Patterns. John Wiley and Sons, New York, U.S.A. (2001).
2. Gamma, E., et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts, U.S.A. (1996).
3. Batteram, H., et al.: AWARENESS Scope and Scenarios. AWARENESS Deliverable D1.1 (2004). Available at http://awareness.freeband.nl.
4. Dockhorn Costa, P., et al.: AWARENESS Services Infrastructure. AWARENESS Deliverable D2.1 (2004). Available at http://awareness.freeband.nl.
5. Dockhorn Costa, P., Ferreira Pires, L., van Sinderen, M., Pereira Filho, J.: Towards a Service Platform for Mobile Context-Aware Applications. In: Mostefaoui, S., K., et al. (eds.): Proc. 1st Intl. Workshop on Ubiquitous Computing (IWUC 2004). Portugal (2004) 48-61.
6. Dockhorn Costa, P., Ferreira Pires, L., van Sinderen, M., Rios, D.: Services Platforms for Context-Aware Applications. In: Markopoulos, P., et al. (eds.): Proc. 2nd European Symp. on Ambient Intelligence (EUSAI 2004). The Netherlands (2004) 363-366.
7. Freeband Kennisimpuls, “AWARENESS project”. The Netherlands (2004). Available at http://awareness.freeband.nl
8. Alexander, C.: The Timeless Way of Building. Oxford University Press (1979).
9. Parlay Group. Parlay X Web Services White Paper (2002). Available at http://www.parlay.org/ about/parlay_x/ParlayX-WhitePaper-1.0.pdf.