Towards a Service Platform for Mobile Context-Aware Applications

Abstract. Context-aware service platforms aim at supporting handling of contextual information in order to provide better user-tailored services. In this paper we propose a novel service platform architecture to support mobile context-aware applications, giving emphasis to the configurability of the platform's generic functionality. We introduce concepts and a language to cope with configurability aspects. We have implemented this architecture in the WASP[1] platform, a web services based context-aware service platform on top of 3G networks.

1 Introduction

Context-awareness has emerged as an important and desirable feature in distributed mobile applications. This feature deals with the ability of applications to utilize information about the user's environment (context) in order to dynamically select and execute relevant services [1] that better match the user needs. In a ubiquitous environment, with many services available at any time, the use of context is especially important to help determining which services are relevant for the user [7].

Building context-aware systems involves the consideration of several new challenges. Such challenges are mainly related to gathering/sensing, modeling, storing, distributing and monitoring contextual information. These challenges motivate the need for proper architectural support.

There have been many initiatives towards defining architectural support for context-aware applications. In particular, considerable effort has been spent on the development of *infrastructure software* to support the development and/or operation of context-aware applications. *Infrastructure software* comprises code libraries or runtime environments that provide higher-level abstractions that shield application developers from the demands of interacting with lower-level data, hardware devices and software concepts [7]. Among these infrastructures, we have seen the emergence of context-aware service platforms, which aim at providing support to application designers to conceive their applications using services, mechanisms and interfaces that shield them from the complexity of handling contextual information [2].

The current platforms, however, offer a limited level of configurability. Ideally, a platform for context-aware applications should facilitate the creation and the dynamic deployment of a large range of context-aware applications that are unanticipated during the design of the platform. In this paper, we define a service platform architecture for context-aware applications, giving emphasis to the configurability and

[1] WASP (Web Architectures for Service Platforms) is a Dutch national project.

extensibility of the platform's generic functionality. We present a descriptive language, coined WSL (WASP Subscription Language), which provides means to applications to dynamically configure the platform. This paper is an extension of [2], in which we identify the essential requirements to be satisfied by a configurable context-aware service platform.

Within the WASP (Web Architectures for Service Platforms) project, we have implemented the proposed service platform architecture. The prototype implementation is referred to as the WASP Platform. The WASP project is concerned with the definition and validation of a service platform to facilitate the development and deployment of mobile context-aware applications, called WASP applications, on top of 3G networks [3], using Web Services [9] technologies. Fig.1 depicts the environment of the WASP platform [11].

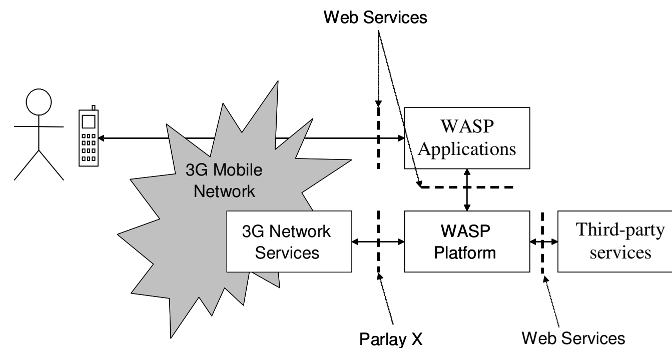

Fig. 1. Overview of the WASP Platform

The 3G networks provide the telecommunication infrastructure for mobile terminals. In addition, a 3G network can play the role of a Context Provider, being able to provide the current location of its users. 3G network functions are accessed using the Parlay X [4], a web services interface. Web Services technologies are used to allow application↔platform and platform↔service provider interactions.

The remainder of this paper is structured as follows. Section 2 gives an overview of the service platform. Section 3 defines the concepts within the platform, and Section 4 describes the WSL. Section 5 introduces the platform components. Section 6 presents the prototype and an example of platform usage, and finally Section 7 concludes the paper, presenting the current project status and some final remarks.

2 Overview of the Service Platform

The service platform forms the system environment for context-aware mobile applications. It supports the system scenario in which context is gathered from *Context Providers* (sensors or third party information providers) and services are implemented by third-party service providers.

The platform aims at delivering the most adequate services based on both application requirements and contextual facts (see Fig. 2).

Applications describe their requirements by defining the desired services and the contextual conditions in which the services should be executed. The platform should autonomously react to reaction rules, in which the contextual conditions are checked against contextual facts.

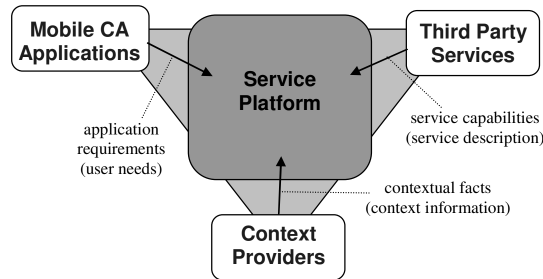

Fig. 2. Overview of the Service Platform

We have identified the essential requirements to be satisfied by the service platform [2]. These requirements include:

- Context Handling: the platform should provide efficient mechanisms to gather, store, distribute and monitor contextual information;
- Reactive Behavior: the platform should allow the specification of reaction rules. Moreover, it should be able to react according to the specified rules;
- Configurability: the platform should be able to support context-aware applications that are unanticipated during platform design. For that, mechanisms of configurability and the use of generic components need to be considered.

The abovementioned requirements are our major concerns in the definition of the platform. We have spent much of our efforts on developing a platform architecture with a high level of configurability. The proposed solution includes the definition of a subscription language, which allows applications to dynamically expose their needs to the platform.

Fig. 3 depicts the proposed service platform architecture, which contains three main modules: *Monitor*, *Registry* and *Context Interpreter*. The *Context Interpreter* gathers contextual information from *Context Providers* (sensors or third-party providers), manipulates contextual information and makes it uniformly available to the rest of the platform. The *Registry* maintains information necessary to support the interpretation of application requirements and the execution of services. The *Monitor* is the core of the platform, since it is responsible for receiving and interpreting application requests making them active within the platform. The subcomponents of the architecture are discussed later in this paper.

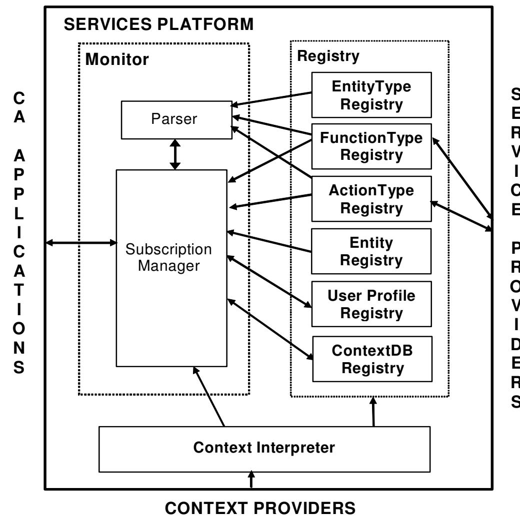

Fig. 3. The Service Platform Architecture

In our approach, application↔platform interactions are dynamically configured through the addition of *application subscriptions*. By means of subscriptions, applications are capable of dynamically exposing their requirements to the platform, which composes at runtime new services from the set of available services. Application subscriptions are written in WSL, a descriptive language we have developed for this purpose.

We have defined a simple extensible context model to explore the contextual knowledge of the service platform.

3 Concepts

In order to effectively and consistently manipulate the contextual knowledge of the platform, we need to organize, represent, and describe it in a model. For this purpose we have introduced the concept of a context model. Once this model is defined, it is used as basis for common understanding between platform, applications and service providers.

In addition, we have defined the types of services that are supported within the platform. The defined service units constitute the building blocks to perform dynamic service composition.

3.1 The Platform Context Model

The service platform manipulates *data entities*, which represent objects of the real world (users, restaurants, museums, roads, vehicles, etc.). *Attributes* (age, area,

address, etc.) and *Context* (time, location, activity, etc.) are associated with data entities.

The UML diagram depicted in Fig. 4 describes a possible configuration of the context model proposed for the platform. This diagram shows only an example configuration, with entities *restaurant* and *user* and their relationship with context *location*. Along the platform usage, the model may be extended by dynamically adding new entities types (e.g., *museum* and *supermarket*) and context types (e.g., *time* and *activity*).

The model presents three instantiation levels, namely a *metamodel*, a *model* and an *object* level. The *metamodel* level is embedded in the platform and it is defined during the platform design-time, being unchangeable during runtime. The *model* and the *object* levels are dynamically changeable during platform runtime. They represent instances of the *metamodel* and the *model* levels, respectively. Details about the context model can be found in [1].

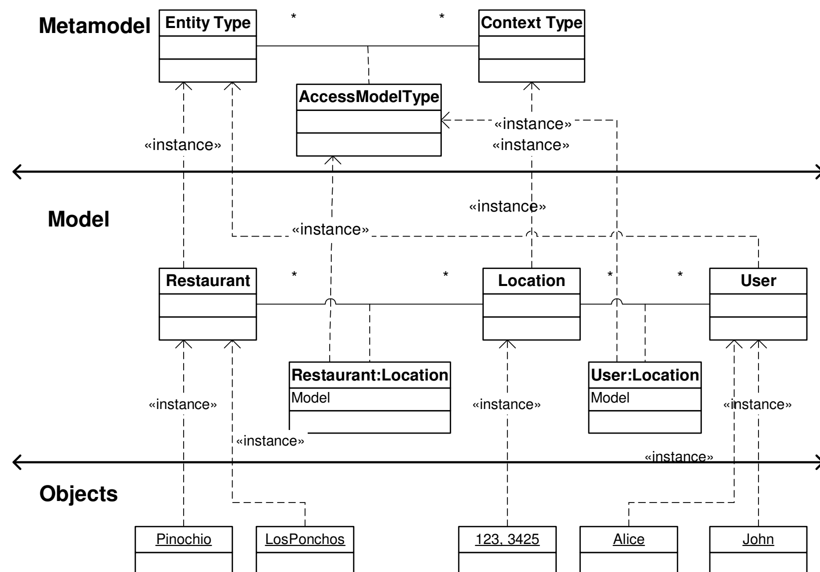

Fig. 4. The platform context model

Fig. 4 presents a model configuration in which entities types *Restaurant* and *User* (*model*) are instances of *Entity Type* (*metamodel*). Moreover, the context *Location* (*model*) is instance of *Context Type* (*metamodel*). Similarly, *Pinochio* and *LosPonchos* (*object*) are instances of entity *Restaurant* (*model*) and *Alice and John* (*object*) are instances of entity *User* (*model*).

It would be possible to define (hierarchical) relations between *entity types*. This could vary from simple categorizations of entity types to complex *ontologies* [5]. A common representation of this knowledge is essential for the interoperability of the platform and its environment.

3.2 Services

We have defined two types of service units within the platform: (i) services used to reason about contextual circumstances, which we call *functions* and (ii) services that correspond to response triggers to determined contextual occurrence, which we call *actions*.

Functions are service units that perform a computation with no side-effects, i.e., it does not change the current status of the platform. An example of function is the `isInside` operation, which may have two parameters, a *container* and a *person*. It returns *true* if one entity type *person* is inside an entity type *container*, and *false* otherwise.

Actions are service units that perform a computation with side-effects for one or more parties involved in the system. An example of *action* is the `sendAmbulance` operation, which sends an ambulance to a physical place. The invocation of functions and actions follows the request-response pattern.

Functions and actions constitute the set of service units that might be composed to form a new service. The service composition is described by applications by means of applications subscriptions written in WSL and performed by the platform.

4 The WASP Subscription Language (WSL)

We have developed a descriptive language to specify subscriptions. Initially, we identified two essential requirements with respect to the elements of this language: (i) a way to specify the reactions of the platform to stimuli from the environment and (ii) a way to correlate events that eventually trigger the specified reactions. In order to fulfill these requirements we have defined the clauses `ACTION` and `GUARD`, respectively.

Subscriptions can be either parameterized or not. Parameterization is necessary when the rule (subscription) applies to a collection of entities, since it would be cumbersome to write a subscription for each target entity. To allow parameterization, the clause `SCOPE` was defined.

Without subscription parameterization, the application would be forced to review all subscriptions that involve the kinds of entities for which the subscription applies. For example, the application would have to add a subscription for each newly introduced entity.

Non-parameterized subscriptions are defined in the form `<ACTION actions GUARD expr>`, where actions represent the reaction of the platform according to stimuli defined in the `GUARD` logical expression.

Sometimes it is necessary to select entities of a collection for which a certain condition holds. For this purpose we have defined the `SELECT` clause. It allows the selection of a subset of a collection respecting the logical combination of entities' contexts and attributes as a condition. The EBNF syntax and the UML class diagram defining the syntax of WSL can be found in [1].

Each clause defined in WSL is discussed in more detail bellow. We discuss the clauses by means of examples that use the functions `count` and `IsInside`, and

the action `sendSms`. `Count` returns the number of elements in a collection; `IsInside` returns *true* if an entity is inside a given *container*, and *false* otherwise; `sendSms` sends a message to one or a set of users.

The SELECT clause

The `SELECT` clause returns a collection of entities for which a given filtering expression holds true. Its abstract syntax is as follows:

```
SELECT (<collection-of-entities>; <var>; <filtering-
expression-involving-var>)
```

A concrete example of the select clause which returns a collection of users currently located in Enschede would be:

```
SELECT (entity.user.*; u; u.location.city ==
"Enschede")
```

`entity.user.*` represents the collection of all users in the system. `u` is the variable to designate the elements of the collection and `u.location.city == "Enschede"` is the logical expression that filters the given collection of users by selecting the ones that are in Enschede.

The ACTION-GUARD clause

The `ACTION-GUARD` clause defines an action (or actions) that should be triggered in consequence of a correlation of events. Its abstract syntax is as follows:

```
ACTION <action> [GUARD <correlation-of-events>]
```

A concrete example is:

```
ACTION    SendSms (entity.user.John, "Hey John, coca-
cola and film, a perfect combination!");

GUARD
 (count (SELECT (entity.cinema.*; c;
           (isInside (entity.user.John, c) AND
           (c.location.city == "Enschede"))
  //list of cinemas, where John is located, inv: 0 or 1
  )>0)
```

This application subscription defines that a message should be sent to John if John is inside a movie theater and this movie theater is located in Enschede. The `SELECT` clause is used to select a collection of movie theaters in Enschede where user John

currently is. The selected collection has 0 or 1 element (either the user is in one or in zero movie theaters). If the user is in one, an advertisement is sent to him, otherwise the action is not triggered.

The SCOPE clause

The SCOPE clause defines a collection of target entities for which the subscription should be applied. An ACTION-GUARD clause is always nested in the SCOPE clause.

The SCOPE clause has the following abstract syntax:

```
SCOPE (<collection-of-entities>; var){
ACTION <action-involving-var> [GUARD <correlation-of-
events>]}
```

An example of usage of the scope clause is the scenario "Send an advertisement to every user inside the movies in Enschede":

```
SCOPE ((SELECT (entity.user.*; u2; u2.location.city ==
"Enschede")); u)
{
 ACTION
    SendSms (u, "Coca-cola and film, a perfect
                 combination!");
 GUARD
     (count (SELECT (entity.cinema.*; c;
                       (isInside(u,c) AND
                       (c.location.city == "Enschede"))
               )>0 )
}
```

As already mentioned, the SELECT clause returns a collection of users located in Enschede in that given moment. The ACTION-GUARD clause is applied for each of the selected users, which are named “u” by the scope clause.

5 Platform Components

The three main components of the platform are the *Context Interpreter*, the *Registry* and the *Monitor*.

5.1 Context Interpreter

The *Context Interpreter* gathers contextual information from *Context Providers* (sensors or third parties providers) and makes it uniformly available to the rest of the platform. The interpreter might also perform:

- *Context aggregation*: the context interpreter provides contextual information about a certain entity by gathering and aggregating context from a set of context providers, if necessary;
- *Context inference*: the context interpreter infers context from other contexts. Inference rules may be used to perform this activity.

5.2 Registry

The *Registry* component is a collection of registries that contain and maintain the information represented in the *data entity model* (Fig. 4). Therefore, they provide essential information to support the deployment of applications in the platform. We have defined six registries:

- Entity Type Registry: stores *entity types*, and their correspondent *attributes* and *context types*. Examples of entity types are person, cinema, restaurant and museum; examples of attributes are age and address; examples of context types are location, velocity and activity. Some types of contexts apply only to specific entity types. For example, velocity may be applied to a person but not to a cinema. The Entity Type registry manages all possible combinations of context types and entity types, being the actual representation of level *Model* of the context model;
- Entity Registry: stores instances of entity types. For example, it might store the instances *Alice* and *John* of the entity type *person,* and *Pizza Hut* of entity type *restaurant*;
- Function and Action Type Registries: store, match and retrieve *functions* and *actions* profiles, respectively. Actions and functions are published and implemented by third-parties service providers;
- User Profile Registry: manages user profiles. Significant facts can be collected directly from the user profiles, like, e.g., as gender, date of birth, name, preferences, etc. This can be considered contextual information in the sense that it describes the environment in which the users operate.
- ContextDB Registry: preserves contextual information over time (history). Keeping context history is essential to allow context inference based on past occurrences.

5.3 Monitor

The core of the platform architecture is the *Monitor* module, which is responsible for interpreting and managing the application subscriptions. In order to perform its operations, the *Monitor* makes use of the data available in the Registry and the contextual information provided by the Context Interpreter.

Application↔platform interactions are dynamically configured through definition of *application subscriptions*. These subscriptions are expressed in WSL, which provides means for applications to dynamically expose their requirements to the platform, allowing the platform to perform a dynamic service composition.

Using WSL one can refer to *entities* (their *context* and *attributes*) and the combination of *actions* and *functions* in order to express the desired service. Since these elements are used through subscriptions and *entities*, *context*, *attributes*, *actions* and *functions* can be added to the platform on demand (during runtime), relatively complex services can be composed and deployed at platform runtime.

The Monitor contains a Parser and a Subscription Manager component.

5.3.1 Parser

The Parser component is responsible for verifying whether the subscriptions are syntactically and semantically correct, having as reference the syntax of WSL [1]. The result of the parsing is a tree containing WSL primitive elements.

There are two levels of semantic checking:

- A model checking level using the platform entity metamodel (Fig. 4). At this level, the Parser verifies the existence of the entity types and the combination and context type and entity type. Furthermore, the Parser needs to verify the semantics of *functions* and *actions*;
- An instance checking level using the instance repositories to check the existence of the entities (final instances).

5.3.2 Subscription Manager (SM)

The SM provides an interface to allow applications to add, remove or update subscriptions. Furthermore, applications should provide a notification interface for possible callbacks from the platform. Fig. 5 depicts the internal configuration of the Subscription Manager.

Applications subscribe to the platform by dynamically adding application subscriptions. Moreover, existing applications subscriptions can be updated or removed.

Added or updated subscriptions are parsed to be verified for syntax and semantic integrity. If no errors are found, they are forwarded to the *event handler*, which constantly checks whether the conditions in the `GUARD` clause has become true or not. The frequency in which verifications are performed depends on how often contextual information is provided by the *Context Interpreter*. In general, the `GUARD` clause is rechecked when notifications of context changes are received from the *Context Interpreter*.

a) Applications interface
b) Parser interface
c) Context Interpreter interface
d) Repositories interfaces
e) Parlay Web Services Interface

Fig. 5. Subscription Manager

When the `GUARD` condition becomes true, an action is triggered, which could be a simple callback to the application or a more complex task on the users' device.

The message sequence diagram depicted in Fig. 6 represents the execution flow of a subscription in the *Subscription Manager*.

Fig. 6. Flow of a subscription in a sequence diagram

In Fig. 6 the user requests a service to the application, which consists of sending him a message in case one of his colleagues is close to him. The application describes the service composition by means of a subscription and adds it to the platform. The Proxy gets the subscription and asks the Parser to check it. Once the subscription is checked, it is forwarded to the EventVerifier. The EventVerifier checks whether there is a colleague close by. In case this condition becomes true, the ActionExecutor triggers the `sendSMS` action.

6 Validation Scenario

We have implemented the proposed service platform architecture within the WASP project. The main goal of the prototype has been to demonstrate and validate the concepts and the main proposed architectural elements, giving emphasis to the interactions between the application and the platform.

6.1 Prototype

We have prototyped the Monitor (*Subscription Manager* and the *Parser*), some of the *Registries* and a simplified *Context Interpreter*. The *application↔platform* and *platform↔context provider* interactions were implemented. The scope of the prototype does not include the implementation of the platform↔service providers interaction. Essential *actions* and *functions* are hard-coded in the platform with which our scenarios could be performed (mainly location-aware scenarios). Future implementations of the prototype will consider the implementation of this interaction by implementing the *action type registry* and *function type registry* using, e.g., the UDDI [10] approach, which allows service discovery.

With respect to the WASP Subscription Language, we have defined an XML Schema that represents the WSL Syntax [1]. Application subscriptions are written by applications in XML structures and validated using the Schema Syntax. The WSL parser is able to read the application subscriptions in XML format and map them into Java classes, which are automatically compiled and executed during runtime.

We have used Web Services technologies and Java language for implementing the prototype. The WASP platform interface is offered as a web service end-point, which allows the operations to be remotely called by the platform applications. Furthermore, we also have implemented the users' terminals as a web service end-point to allow callbacks from the platform. We have used JAX-RPC [6] to automatically generate the WSDL file from Java interfaces. We have used the W3C's Document Object Model (DOM) [8] to parser application subscriptions written in XML format.

6.2 Example of Platform Usage

In this example, a context-aware taxicab application is deployed on top of the platform. Users make requests for a taxicab in the company's web site with no need to inform their current location. Furthermore, users get a message when the taxicab approaches their location. Fig. 7 depicts the sequence diagram of this scenario and Table 1 presents the respective interaction messages.

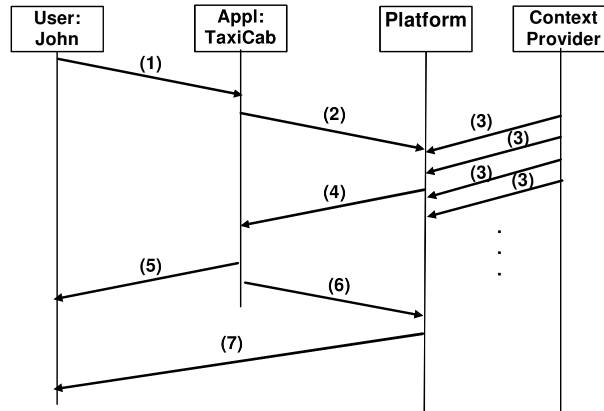

Fig. 7. Sequence diagram for the taxicab scenario

Table 1. Exchanged messages in the taxicab scenario

Message Number	Message Contents
(1)	"I need a cab"
(2)	`ACTION NotifyApp( bookTaxicab (SELECT (entity.taxicab.*; tc; (CloseBy (tc, entity.user.John, 3000)) AND (tc.company = "ABC"))));`
(3)	John's location and taxicabs' locations
(4)	The booked taxicab identification and the approximate taxicab arrival time
(5)	"The taxicab will approximately arrive 5 minutes"
(6)	`ACTION SendSMS(entity.user.John, "Your taxicab has arrived."); GUARD CloseBy(entity.user.John, entity.taxicab.cab1234, 50)`
(7)	SMS with the text "your taxi has arrived"

In Table 1, message (1) represents the user request for a taxicab. Message (2) is the application subscription in WSL for a composed service whose service units are actions `NotifyApp` and `bookTaxicab`, and function `closeby`. The action `NotifyApp` performs a callback from the platform to the application, to inform the results of action `bookTaxicab`. These results are the identification of the taxicab

and an approximate arrival time. Besides returning this information, `bookTaxicab` is a third-party service that books a taxicab from a given collection. We assume for this scenario that the context provider is able of providing the current taxicabs locations and the users locations (message (3)).

An extra service offered by this context-aware taxicab application is that the user gets a message when the taxicab approaches his current location. Message (6) represents the application subscription in WSL used to perform this operation. The `ACTION-GUARD` clause defines that a message should be sent to the user when the specific taxicab is close to him.

Other example scenarios of platform usage, including a policemen application and a follow me application, have been reported [1].

7 Final Remarks

This paper investigates some of the technical challenges related to the design of context-aware service platforms, and proposes a generic platform architecture for supporting the development of context-aware applications to tackle the identified issues.

Most of the current approaches for building context-aware service platforms do not consider the dynamic deployment of mobile context-aware applications on top of a service platform. For this reason, we have explored this aspect of the proposed architecture in more detail. Our approach provides means to configure interactions between applications and platform at runtime. Furthermore, the platform may be extended through the addition of *functions*, *actions* and *data entities*. Embedding this level of flexibility in the platform makes it appropriate for a large range of (unanticipated) context-aware applications.

In order to allow dynamic configuration of applications↔platform interactions, the proposed approach makes use of a descriptive language. This language, called WSL, is used to specify how the platform must react to a given correlation of events, potentially involving contextual information.

We have used *Web Services* as the technology to enable the interactions of the platform with its environment. As a consequence, third-party applications can access the services offered by the platform through widely-used Internet protocols. In addition, Web Services facilitate the extension of the platform by third-parties, which may provide additional *functions* and *actions* as Web Services.

Defining a complete architecture for a context-aware service platform is a non-trivial task. It involves several issues related to different domains, such as ubiquitous computing, artificial intelligence, human-computer interaction, and other crosscutting issues such as security and privacy, scalability and performance.

Within the WASP project, there has also been an effort to investigate the applicability and usefulness of Semantic Web technologies for the representation of contextual information, leading to the development of context ontologies and the use of reasoners to detect context conditions [5]. This work advocates that the use of ontologies, in combination with the presented architecture, enhancing the flexibility, reusability and reasoning capabilities of the platform.

An additional effort within the WASP project [11] aims at designing a privacy architecture for the WASP platform, providing users of the platform with control over their privacy while being unobtrusive. This effort proposes an approach for privacy control based on P3P (Platform for Privacy Preferences Project) policy language.

Further developments of the platform will consider the integration of the efforts within the WASP project, giving special attention to the use of Semantic Web technologies. We believe that the use of ontologies and reasoners are promising techniques to allow reusability, flexibility and more intelligent behavior of context-aware systems.

Acknowledgements

The work described in this paper has been sponsored by *Freeband Knowledge Impulse*, a joint initiative of Dutch Government, knowledge institutes and industry.

References

[1] Dockhorn Costa, P., *Towards a Services Platform for Context-Aware Applications*. Master Thesis, University of Twente, The Netherlands, August 2003.

[2] Dockhorn Costa, P., et al., Architectural Requirements for Building Context-Aware Services Platforms. *Proc. of 9th Open European Summer School and IFIP Workshop on Next Generation Networks (EUNICE 2003)*, Hungary, September 2003.

[3] Laar, V., Requirements for the 3G Platform. *WASP Deliverable: D1.1*, January 2003.

[4] Parlay X Web Services White Paper. *The Parlay Group white paper*, December 2002. [http://www.parlay.org/ about/parlay_x/ParlayX-WhitePaper-1.0.pdf].

[5] Rios, D., et al., Using Ontologies for Modeling Context-Aware Services Platforms. *Workshop on Ontologies to Complement Software Architectures (OOPSLA 2003)*, Anaheim, CA, October 26-30, 2003.

[6] Sun Microsystems, Java API for XML-Based RPC (JAX-RPC) Specification 1.0, JSR-101. [http://java.sun.com/xml/downloads/jaxrpc.html#jaxrpcspec1].

[7] W. Keith Edwards, et al., Stuck in the Middle: The Challenges of User-Centered Design and Evaluation for Infrastructure. *Proc. of the Conference on Human Factors in Computing Systems (CHI 2003)*, Fort Lauderdale, FL, April 5-10, 2003.

[8] World Wide Web Consortium. Document Object Model (DOM) Level 1 Specification. October, 1998. [http://www.w3.org/TR/REC-DOM-Level-1/].

[9] World Wide Web Consortium. Web Services Architecture. August, 2003. [http://www.w3.org/TR/ws-arch/]

[10] Universal Description, Discovery and Integration (UDDI) project. *UDDI: Specifications*. [http://www.uddi.org/specification.html].

[11] Zuidweg, M., Using P3P in a Web Services-Based Context-Aware Application Platform. *Proc. of 9th Open European Summer School and IFIP Workshop on Next Generation Networks (EUNICE 2003)*, Hungary, September 2003.