



Towards a Services Platform for Context-Aware Applications

Patrícia Dockhorn Costa

**Thesis for a Master of Science degree in
Telematics from the University of Twente,
Enschede, The Netherlands**

Graduation Committee:

Dr. José Gonçalves Pereira Filho

Dr. ir. M. J. van Sinderen

Dr. L. Ferreira Pires

Enschede, The Netherlands

August, 2003

Abstract

Context-aware computing deals with the ability of computer systems to take advantage of information from or conditions in the dynamic environment to provide added-value services or to execute more complex tasks. In addition to dealing with explicit input, context-aware applications consider contextual information (implicit input).

Context awareness has received attention in recent years with the development of mobile computing and the appearance of a new generation of mobile devices.

Building context-aware applications is challenging due to the fact that contextual information is being manipulated. The challenges related to gathering/sensing, modeling, storing, distributing and monitoring context justify the need for proper architectural support.

This thesis proposes a generic and configurable services platform architecture to support context-aware applications. The services platform aims at providing support for application designers to conceive their applications using services, mechanisms and interfaces that shield them from the complexity introduced by handling contextual information.

A distinctive characteristic of our platform is that it enables the dynamic deployment of a large range of context-aware applications that are unanticipated during the design of the platform. We define a subscription language that applications use to configure the platform to react to a given correlation of events, potentially involving contextual information. In addition, we use Web Services as a technology to enable the interactions of the platform with its environment.

Table of contents

1 INTRODUCTION	1
1.1 MOTIVATION	1
1.2 OBJECTIVES	4
1.3 APPROACH	4
1.4 STRUCTURE	5
2 CONTEXT AWARENESS	6
2.1 CONTEXT	6
2.1.1 What is context?	6
2.1.2 What is context-awareness?	7
2.2 GATHERING CONTEXT	8
2.3 MODELING CONTEXT	8
2.3.1 Nature of contextual information	9
2.3.2 Primitive contexts	10
2.3.3 Examples of Techniques to Model Context	11
3 CURRENT RESEARCH EFFORTS IN CONTEXT-AWARE COMPUTING.....	14
3.1 CONCEPTUAL FRAMEWORKS	14
3.1.1 Cooltown Project	14
3.1.2 Context Toolkit	15
3.2 SERVICE PLATFORMS	16
3.2.1 Platform for Adaptive Applications	17
3.2.2 M3 Architecture	18
3.3 APPLIANCE ENVIRONMENTS	18
3.3.1 A Universal Information Appliance (UIA)	18
3.3.2 Ektara Architecture	19
3.4 COMPUTING ENVIRONMENTS	20
3.4.1 Portolano	20
3.4.2 PIMA	21
3.5 DISCUSSION	22
4 THE WASP PLATFORM REQUIREMENTS.....	24
4.1 CONTEXTUAL INFORMATION	24
4.1.1 Context representation/modeling	24
4.1.2 Context storage/retrieval	25
4.2 PLATFORM INTERACTIONS	25
4.2.1 Support for different kinds of context providers	26
4.2.2 Reactive behavior	27
4.2.3 Coordination among different applications	29
4.2.4 Discovery and publishing of services	30
4.3 GENERAL REQUIREMENTS	31
4.3.1 Support for security and privacy services	31
4.3.2 Charging	31
4.4 CONCLUDING REMARKS	32
5 THE WASP PLATFORM ARCHITECTURE DESIGN.....	34
5.1 OVERVIEW OF THE ARCHITECTURE	34
5.2 THE WASP PLATFORM CONTEXT MODEL	36
5.3 CONTEXT INTERPRETER	38

5.3.1	Information Provisioning Models of the Context Interpreter	40
5.3.2	Inferring Context	41
5.4	REPOSITORIES	43
5.4.1	Entity Type Registry	43
5.4.2	Function Type Registry	43
5.4.3	Action Type Registry	45
5.4.4	Service Registry	48
5.4.5	Entity Registry	49
5.4.6	User Profile Registry	49
5.4.7	ContextDB Registry	50
5.5	MONITOR	51
5.5.1	The WASP Subscription Language (WSL)	52
5.5.2	The Subscription State Machine	57
5.5.3	Scenarios	58
5.5.4	Parser	63
5.5.5	Subscription Manager (SM)	64
5.5.6	Coordinator	67
6	IMPLEMENTATION	69
6.1	APPROACH	69
6.2	APPLICATION-PLATFORM INTERACTION	71
6.2.1	WASP Application (client side)	73
6.2.2	WASP Platform (server side)	74
6.2.3	The WSL Parser	74
6.2.4	Web Services – Java implementation issues	74
6.3	PLATFORM-CONTEXT PROVIDER INTERACTION	74
6.3.1	Context Provider (server side)	74
6.3.2	WASP Platform (client side)	74
6.4	SCENARIOS	74
6.4.1	Policemen Scenario	74
6.4.2	Advertisement Scenario	74
6.4.3	Proximity Scenario	74
6.5	CONCLUDING REMARKS	74
7	CONCLUSIONS.....	74
7.1	GENERAL CONCLUSIONS	74
7.2	FUTURE WORK	74
	REFERENCES.....	74
	APPENDIX A WSL - XML SCHEMA.....	74
	APPENDIX B SUBSCRIPTION INTERFACE - WSDL	74
	APPENDIX C USER LOCATION SERVICE - WSDL	74
	APPENDIX D POLICEMEN SCENARIO (WSL-XML).....	74
	APPENDIX E ADVERTISEMENT SCENARIO (WSL-XML)	74
	APPENDIX F PROXIMITY SCENARIO (WSL-XML).....	74

Preface

This thesis describes the results of a Master of Science assignment at the Architecture of Distributed Systems Group at the University of Twente. This assignment has been carried out from December 2002 to July 2003 as part of the WASP (Web Architectures for Services Platforms) project.

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank the Architecture of Distributed Systems Group for granting me with the opportunity to do the Master of Science in Telematics at the University of Twente. I furthermore would like to thank my supervisors José Gonçalves Pereira Filho, Marten van Sinderen and Luís Ferreira Pires whose help, stimulating suggestions and encouragement helped me throughout the development of this work.

I would also like to thank the colleagues at the Arch Lab for the pleasant working environment and all friends for making my staying in the Netherlands very joyful. In particular, I am very grateful to my family in Brazil for their fundamental emotional support.

Especially, I would like to give my thanks to my boyfriend and future husband João Paulo Andrade Almeida whose patient love enabled me to complete this assignment.

Enschede, 22nd July 2003.

Patrícia Dockhon Costa.

1 Introduction

This chapter presents the motivation, the objectives, and the structure of this thesis. It identifies the relevance of context-aware computing and draws special attention to the support of services platforms in the context-aware computing domain.

This chapter is further structured as follows: Section 1.1 briefly presents the motivation of this work, Section 1.2 states the objectives of this thesis, Section 1.3 presents the approach adopted in the development of this thesis and Section 1.4 outlines the structure of this thesis by presenting an overview of the chapters.

1.1 Motivation

Computing is moving from the traditional desktop paradigm to a mobile computing paradigm, in which new types of computing devices augment the users' workspace and the user environment changes dynamically as a consequence of the user's mobility.

This new paradigm has brought the possibility of exploring the dynamic context of the user. However, most computer systems are still designed to ignore (or assume fixed) contextual information and process their work based only on explicit input (Figure 1). Therefore, these systems do not take advantage of information from or conditions in the dynamic environment in order to provide added-value services or to execute more complex tasks [30].



Figure 1 - Traditional applications

Context-aware computing deals with the ability of computer systems to obtain contextual knowledge in order to perform improved services. Rather than treating mobility as a problem to be solved, context-aware computing seeks to exploit the

nature of it. As a consequence, it creates a new generation of applications in which the user-application interaction is enhanced by perceiving/sensing the surrounding environment. Contextual information of what, when and where the user is, what the user knows and what the user and system capabilities are, can greatly simplify the user scenario. Such manipulation of contextual information can also be used to reduce the teaching needed for the user accomplishment of tasks. Context-aware applications consider contextual information in addition on explicit input, as illustrated in Figure 2. This contextual information is implicitly gathered from the applications' environment.

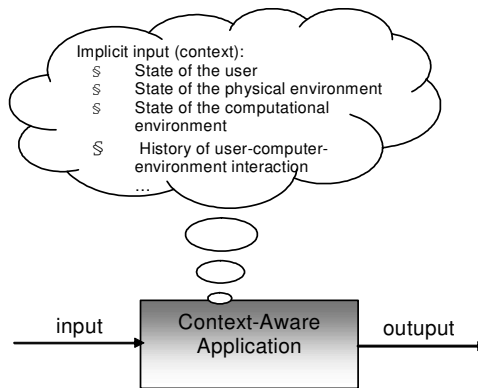


Figure 2 - Context-Aware Applications

Context has been object of studies in different fields of computer science. As a consequence, several definitions for context have been put forward in the literature, serving different purposes. In Information Bases, for example, context describes a group of conceptual entities from a particular standpoint [43]. In Artificial Intelligence, context appears as means of partitioning a knowledge base into manageable sets or as logical construct that facilitates reasoning activities [32, 17].

In context-aware computing domain, important aspects of context that have been initially explored by its research community were mostly related to *where you are*, *who you are with*, and *what resources are nearby* [40]. In this thesis, we have embraced the following informal definition of context, which has been used as a reference in the literature of context-aware computing domain [10].

“Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.”

Context awareness is being explored in several research efforts, particularly in research communities such as Ubiquitous Computing and Human Computer Interaction (HCI). This subject has received attention in recent years, with the development of mobile computing and the appearance of a new generation of mobile devices. Since then, a number of context-aware systems have been proposed, ranging from specific context-aware applications to generic support platforms.

Building context-aware applications involves the consideration of several new challenges. Such challenges are related to gathering/sensing, modeling, storing,

distributing and monitoring contextual information. These challenges justify the need for proper architectural support. Not surprisingly, this has been subject of many projects at research centers such as the Lancaster University [14], IBM [15], MIT [8], Hewlett-Packard [26] and the University of Queensland [2], each project tackling specific problems and aiming at specific characteristics, as will be explored in Chapter 3 of this thesis.

Some of these initiatives propose the development of services platforms for context-aware applications. Services platforms aim at providing support to application designers to conceive their applications using services, mechanisms and interfaces that shield them from the complexity introduced by handling contextual information.

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. This specific issue is one of the main concerns of this work.

Examples of applications that could profit from a flexible services platform are:

- Reminder applications that allow users to set reminders to be triggered according to the occurrence of (a combination of) events;
- Advertisement applications that allow third parties to advertise to users according to the occurrence of (a combination of) events;
- Medical applications that allow sending help to a user after detecting medical emergencies;
- Security applications that allow policemen to be constantly informed (on a map) about the location of other policemen (colleagues) around them;
- Parking place applications that allow a user to gather information about the closest parking places around him/her. Such information can be number of available parking places, opening/closing times, costs and payment systems, the route (showed on a map) to get to the parking places' locations;
- Bus applications that allow users to keep track of the buses in service. Therefore, it is possible to know when a bus is approaching the bus stop;
- Redirecting applications that allow users to enable the automatic redirection of calls to their work places or homes (depending on their current location);
- Taxicab applications in which users ask for a taxicab without the need to specify their current location. Moreover, users can be informed when the taxicab is approaching;
- Proximity applications that allow users to be informed when things (objects, buildings, restaurants, etc.) or other users are close to them.

Without a configurable and generic services platform, it may be impossible to dynamically deploy such a large range of distinct context-aware applications or this deployment may demand too much effort from application developers.

1.2 Objectives

The main goal of this thesis is to define a configurable services platform architecture to support context-aware applications.

This work has been executed inside the WASP project [46]. In this sense, this work aims at defining a high-level architecture of the WASP Platform. The WASP project is concerned with the definition and validation of a services platform to facilitate the development and deployment of context-aware applications (called WASP applications) on top of 3G networks [27], using Web Services [1] infrastructures. Figure 3 depicts a high level view of the WASP Platform, as foreseen in the original project description.

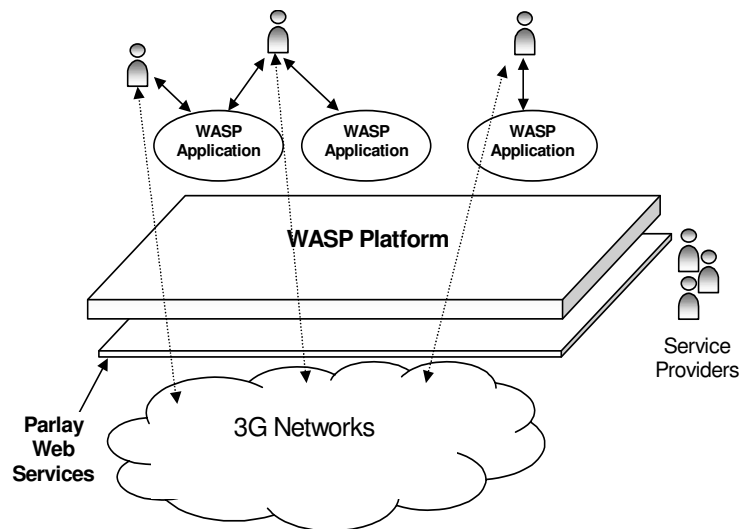


Figure 3- WASP platform

The WASP platform offers business opportunities to service providers that want to expose their services to the users of the platform. Examples of services providers are hospitals, restaurants, museums, etc. WASP applications are intended to be used by mobile users and for this reason, the WASP Platform makes extensive use of the capabilities offered by the underlying network, in this case 3G Mobile Networks.

1.3 Approach

Our efforts towards the definition of the WASP platform architecture include:

- The investigation of essential user and application' requirements for building a context-aware services platform.; The platform should be able to provide services for a large number of new and existing context-aware applications;

-
- The derivation of architectural elements for some of the identified requirements;
 - The integration of the identified architectural elements forming the platform architecture, whose functionality should be sufficient to provide the expected services; and,
 - The development of a prototype for the purpose of demonstration and validation of such an architecture.

In our proposed architecture, interactions between WASP applications and the WASP platform are configured during the platform run-time through the addition of *application subscriptions*. Application subscriptions are written in a descriptive language that allows applications to dynamically expose their needs to the platform. With this language, it is possible to manipulate the representation of the entities involved in the system (users, museums, restaurants, hospitals, vehicles, etc), their attributes, and their context. For instance, it is possible to express that an action involving an entity's context (send an ambulance to John's location) should be taken if an entity (John) enters in a certain context (having a heart attack or a stroke).

1.4 Structure

The structure of this thesis reflects the order in which these issues have been dealt with throughout the research process. This thesis is structured as follows:

- Chapter 2 reports on the conducted literature study, presenting basic concepts in context-aware computing. Issues such as the *nature of contextual information* are addressed in this chapter;
- Chapter 3 presents several research projects that explore different methodologies/approaches for developing context-aware systems showing the contrast of those methodologies/approaches in terms of their main characteristics;
- Chapter 4 identifies the essential requirements to be satisfied by the WASP platform. The issues discussed in this chapter are used as a reference for the architecture conceptual design phase
- Chapter 5 describes the design of the WASP platform architecture proposed by this work. The chapter places special attention to the applications-platform interaction. A subscription language (coined WASP Subscription Language, WSL) is detailed in terms of its clauses, syntax and semantics. Scenarios of platform usage are also presented in this chapter;
- Chapter 6 describes the implementation of selected modules of the proposed architecture. The prototype allows demonstrating and validating the main proposed architectural elements. Examples of demonstrative scenarios are also shown in this chapter;
- Finally, Chapter 7 presents our final conclusions, important remarks and indicates topics for future work.

2 Context Awareness

This chapter reports on the conducted literature study, presenting basic concepts in context-aware computing. Issues such as *the definition of the notion of context* and the *nature of contextual information* are addressed. Furthermore, this chapter discusses two of the major issues related to the manipulation of contextual information: *Gathering Context* and *Modeling Context*.

This chapter is further structured as follows: Section 2.1 presents definitions for context and context-awareness. Sections 2.2 and 2.3 elaborate on the *Gathering Context* and *Modeling Context* issues, respectively.

2.1 Context

The use of contextual information is essential to explore the possibilities of context-aware computing. Nevertheless, while it is simple to form an intuitive notion of context, elucidating a precise definition of it is challenging [10]. The next two sections discuss some of the existing definitions for context and context-awareness, from the perspective of ubiquitous computing. Results from the AI research community, which analyses context in a more formal setting, are not discussed here. However, it is important to mention that some of these results can be used as a basis for achieving a common semantic understanding of contextual information as well as to develop reasoning mechanisms in context-aware platforms.

2.1.1 What is context?

Several definitions for context have been put forward in the literature, serving different purposes. In Information Bases, for example, context describes a group of conceptual entities from a particular standpoint [43]. In Artificial Intelligence, context appears as means of partitioning a knowledge base into manageable sets or as logical construct that facilitates reasoning activities [32, 17].

Although context has already been subject of investigation in different fields, only recently this notion has been explored for ubiquitous computing. Most of the

initial efforts for defining context in ubiquitous computing were specific for certain kinds of context - location and time being the more obvious examples. Schilit and Theimer [40], in 1994, claimed that the important aspects of context were the user location and identities of nearby people. Brown et al. [4] and Ryan et al. [39] gave their definition in terms of examples of context information instead of generalizing the concept. Since the number of examples that can be given is limited, the application of this definition is also limited.

Schilit et al. claim that the important aspects of context are *where you are*, *who you are with*, and *what resources are nearby*. They define context to be the changing environment and the environment is composed by the following views:

- Computing environment: e.g., available processors, devices accessible for user input and display, network capacity, connectivity, and costs of computing;
- User environment: e.g., location, collection of nearby people, and social situation; and,
- Physical environment: e.g., lighting and noise level.

Also this definition turned out to be too specific. It was necessary to give definitions without having to enumerate examples of context because the user experience changes from situation to situation. For those reasons, Dey and Abowd [9] came up with a more generic definition of context, which is "*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves*". In this thesis, we adopt this informal definition as a reference.

2.1.2 What is context-awareness?

Context-awareness seeks to exploit human-computer interactions by providing computing devices with knowledge of the users' environment, i.e., with context. Awareness of the context can potentially be used to diminish the amount of explicit input a user is required to give to a computing system. Contextual information of what, when and where the user task is, what the user knows and what the user and system capabilities are, can greatly simplify the user scenario. Such manipulation of contextual information can also be used to reduce the teaching needed for the user accomplishment of tasks.

A context-aware application definition was given by Schilit and Theimer [40], in 1994, as *software that adapts according to its location of use, collection of nearby people and objects, as well as changes to those objects over time*. Since then, the definitions of context-aware applications were related to applications' adaptation, reactivity, responsiveness and sensitiveness to context. For instance, Pascoe et al. [36] define context-aware computing to be the ability of computing devices to detect and sense (sensitiveness), interpret and respond to (reactivity) aspects of a user's location environment and computing devices. In [4, 40], context-aware applications are defined as applications that dynamically change or adapt their behavior based on the context of the application and the user. Fickas et al. [16] define context-aware applications (called environment-direct applications) to be applications that monitor changes in the environment and adapt their operation according to predefined or user-defined guidelines. Finally, Dey et al. [10] claim it

was necessary to give a more generic definition, which is not bound to a specific characteristic (adaptation, reactivity, responsiveness or sensitiveness). Dey's definition states that a *system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task*. According to this definition, to be a context-aware application, the only requirement is to respond to context and thus, detection, interpretation and adaptation are not mandatory characteristics. This definition makes sense, for instance, in applications that do not adapt to context but simply reflect the context to the user or the ones that do not detect or sense context (detection and interpretation can be performed by other computing entities). We consider this informal definition of context-awareness as a reference for our work.

2.2 Gathering Context

Gathering Context includes three basic sets of activities: *sensing the low-level context*, *building higher-level context* and *sensing changes in the context*. Sensing the low-level context refers to techniques to measure context information from physical sensors, for example, measuring the location of a user and the time or temperature of a place. Building higher-level context refers to refinement activities (context aggregation and interpretation), for instance, location in form of latitude and longitude can be associated with a specific street or building. Aggregation of different contextual information, for example, could be the combination of current time, time zone of the user and his schedule might deliver a hint of the user's current activity. Sensing context changes refers to the constant notification of context changes, for example, the location of a user constantly changes and must be continuously measured and provided.

The quality of the context-aware services directly depends on the quality of the information gathered from the context sources. For this reason, handling context activities should be carefully explored in context-aware systems.

In order to provide a complete infrastructure covering most of the relevant contextual information, context sources could come together to create a net of cooperation in which one context source completes the information provided by the other. Buchholz [5] suggests the agglomeration of context sources provides services, designated by him as *Context Information Service (CIS)*. CIS is a service that finds, measures, interprets and aggregates the relations between context information provided by different context sources. CIS decouples the tasks of gathering, interpreting and aggregating context information from the manipulation of context information for a specific purpose. Therefore, CIS (and cooperation between CIS's) alleviates context-aware systems from the burden of sensing the context from sensors, whose locations and communication protocols are not known in advance. Buchholz also suggests that a new business role, called *Context Provider*, will be the one responsible for building and maintaining the cooperation between CIS's.

Details about *Gathering Context* activities can be found in [37].

2.3 Modeling Context

Context-aware systems based on ad hoc models of context lack in flexibility and expressiveness to model contextual information. Therefore, researchers in ubiquitous computing are looking for systematic approaches and models that

enable capturing the dynamic nature of contextual information, allowing better expression of their main characteristics and complex interrelationships.

In the following sections, we discuss some issues related to context modeling. We will present two fundamental aspects that should be considered when defining a context model: *nature of contextual information* and *primitives contexts*. Furthermore, we will briefly present two example techniques to model context: the W3C approach (*CC/PP*) and *Ontology-based* approaches.

2.3.1 *Nature of contextual information*

According to [21] there are a number of observations about the nature of contextual information, which determine the design requirements for the intended model of context:

- *Context information exhibits a range of temporal characteristics.* Context can be classified as *static* and *dynamic*. Static context information describes invariant aspects such as person's date of birth. The level of dynamism influences the means by which context information must be gathered. Frequently context-aware applications are interested in more than the current state of the context. Accordingly, context histories will frequently form part of the context description.
- *Context information is imperfect.* Ubiquitous computing environments are highly dynamic, which means that information describing them can quickly become out of date. Frequently, the sources, repositories and consumers of context are distributed and information supplied by producers requires processing in order to transform it into the form required by consumer. These factors can lead to large delays between the production and the use of context information. Another problem is that context producers, such as sensors, derivation algorithms and users may provide faulty information. This is particularly a problem when the context must be inferred from crude input provided by sensors. For example, when person's activity must be inferred indirectly from other context information such as location and sound level.
- *Context information is highly interrelated.* Several relationships are evident between contextual information (for example, proximity between users and their devices). Other less obvious types of relationships also exist among context information. Context information can be related by derivation rules which describe how information is obtained from one or more pieces of information. Modeling concepts should exist to express this fact.
- *Context has many alternative representations.* Much of the context information involved in ubiquitous systems is derived from sensors. There is usually a significant gap between sensor output and the level of information that is useful to applications, and this gap must be bridged by various kinds of processing of context information. Moreover, requirements can vary between applications. Therefore, a context model must support multiple representations of the same context in different forms and at different levels of abstraction, and must be able to capture the relationships that exist between the alternative representations.

Considering these intrinsic features of contextual information is essential when defining a context model approach. It is argued that some of these features cannot be easily described using traditional techniques from information systems field, such as ER or UML diagrams; accordingly special constructors should be provided by the modeling approach. Some authors claim that commitment to an *ontology* [41] forms the first step toward implementing this model within a knowledge representation technology. Section 2.3.3 discusses ontology-based approaches to model context.

Ideally, an architecture for context-aware applications should explore a complete context model to support manipulation of contextual information. However, for practical reasons, we have addressed a simple context model that supports the minimum requirements needed for context representation in the scope of this project.

2.3.2 *Primitive contexts*

Apart from the philosophical discussion on the semantic of context there are some primitive forms of context taken from the user's environment that designers agree on. Examples are *location*, *user*, *time*, and *object/device*. We briefly comment on the modeling requirements for these contextual factors in the following paragraphs.

Location

Location can be described in different ways, depending on the application requirements. From the modeling point-of-view, the user can choose to use absolute *versus* relative location, location point *versus* location area, fixed location *versus* moving location, according to the desired level of abstraction. Current location models can be grouped in two categories, *physical* and *geographical* models [11]. Physical location is related to a global geographic coordinate system and provides an absolute, accurate, grid based position in a form of a <latitude, longitude> pair. Geographical location is used to deal with natural geographic objects, such as countries, cities, and also zip codes, postal addresses and so on. The remarkable property of such objects is their clear hierarchical organization. Such a position description is suitable for delivery of spatial information to a human. It is important that the interpretation of these representations is unique, and that appropriate transformations are supported, i.e., the parties using this representation must agree upon the semantics of the location information.

User

In general, data about the user are stored in the so-called *user profile* (e.g., student profile). Significant facts can be collected directly from the user profiles. Knowing these facts enable applications to adapt to their users in many different ways, and to set different system behaviors. The information contained in the profile can be considered as contextual information in the sense that it describes the environment in which the users desire to operate. As such, it represents just a small part of the information domain of context-aware systems. Typically, a user profile consists of the conceptual building blocks Identities (e.g. user Id, password), Characteristics (e.g., place of birth, gender), Preferences (e.g., background color, food preferences), Interests (e.g., opera music, sports) and History (e.g., log of actions). From the modeling point-of-view, what we need is a generic way for describing metadata about the users.

Time

In handling a dynamic environment such as ubiquitous computing, the model needs to support inferences on how objects have changed over time and predictions on how they may change in the future. Also, different aspects of time should be considered by the model (e.g., time point *versus* time interval, time zones, schedules, etc.).

Object/device

For context-aware services, it is important to know which objects/devices are in the user's environment. Examples of physical objects are buildings, busses, trains, shops and signs. Several of these objects are services providers. An example is a hotel that provides services, such as reserving a room, having a public toilet, offering food, etc. It is necessary to have a common vocabulary that allows us to describe objects and device capabilities. Some of the existing approaches are based on the use of the Resource Description Framework (RDF), which is a technique for representing knowledge [47]. For example, the Composite Capabilities/ Preference Profiles (CC/PP) technique, described in the next section.

The architecture proposed by this thesis supports a simple representation of all the abovementioned primitive contexts. *Location, users, time* and *objects/devices* are represented in a model, called the *entity type model*, which defines their main characteristics and their relationship (Section 5.2). Our intention, in the first moment, is to have a simple context representation, which is sufficient for the platform operation. A more elaborated context representation, for example using the techniques indicated in Section 2.3.3, is indicated for future work.

2.3.3 *Examples of Techniques to Model Context*

W3C Approach (CC/PP)

CC/PP (*Composite Capabilities/Preferences Profiles*) is a W3C proposed standard for describing device capabilities and user preferences. Although the current focus is on wireless devices such as PDAs and mobile phones, CC/PP is designed to work with a wider variety of web-enabled devices. The proposal also intends to support applications such as browsers, email, calendars, etc.

In technical terms, CC/PP is an RDF (Resource Description Framework)-based framework for describing and managing software and hardware profiles that include information on the user's device capabilities (physical and programmatic), the user's specified preferences within the user agent's set of options, and specific qualities about the user agent that can affect content processing and display, such as physical location. RDF [47] is a framework for describing metadata. RDF can be serialized in many different ways, and CC/PP uses the XML serialization. An example of location information modeling in CC/PP is showed below. The specification shows an extension of CC/PP to structure a Location profile that contains four components (PhysicalLocation, LogicalLocation, GeodeticLocation and Orientation) [24]:

```
[LocationProfile
  [PhysicalLocation [Country, State, City, Suburb]]
  [LogicalLocation [IPAddress]]
  [GeodeticLocation [Longitude, Latitude, Altitude]]
  [Orientation [Heading, Pitch]]
]
```

There are, however, limitations in CC/PP which make this model not very suitable as a context model for future ubiquitous systems. According to [19], it becomes difficult and unintuitive to use CC/PP when the relationships and constraints in the context model are complex. A novel representation format called Comprehensive Structured Context Profiles (CSCP) has been developed by [19] and it is claimed to overcome the shortcomings of the CC/PP specification language regarding structuring.

Complex ubiquitous systems require much more sophisticated context models in order to support seamless adaptation to changes in the computational environment. The context models will need to specify a range of characteristics of context information, including temporal characteristics (freshness and histories), accuracy, resolution (granularity), confidence in correctness of context information, as well as a variety of information types (including various types of dependencies).

Ontology-based approaches

Ontologies are believed to be a key requirement for building context-aware systems [6] because (i) a common ontology enables knowledge sharing in a distributed system, (ii) ontologies with well defined declarative semantics can be used by different systems to reason about contextual information and (iii) explicitly represented ontologies allow devices and computer applications to interoperate.

Using ontologies, context-aware services are able to (semi)automatically matching users with content/services that are relevant to their context, such as locations, activities and other contextual attributes. Examples of such services could be context-aware message filtering, context-aware travel planning, context-aware restaurant concierge, context-aware notification agents, etc.

An example of a context-oriented ontology approach being developed is CoOL (Context Ontology Language), described in [41]. The code below shows an example of the usage of the terminology introduced by CoOL, where a specific context information (geographical position) with respect to a specific aspect (Gauss-Krueger coordinates) characterizing a specific entity (mobile phone) is expressed in an XML instance document:

```
<instance
xmlns="http://demo.heywow.com/schema/cool"
xmlns:a="http://demo.heywow.com/schema/aspects"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<contextInformation>
  <entity system="urn:phonenumber">
    +49-179-1234567
  </entity>
  <characterizedBy>
    <aspect name="GaussKruegerCoordinate">
      <observedState xsi:type="a:o2GaussKruegerType">
        367032533074
      </observedState>
      <units>10m</units>
    </aspect>
    <certaintyOfObserver>90</certaintyOfObserver>
  </characterizedBy>
</contextInformation>
</instance>
```

3 *Current Research Efforts in Context-Aware Computing*

The goal of this chapter is to discuss what has been proposed in the literature in terms of methodologies and approaches supporting context-aware computing, using example projects as basis for the discussion. The set of projects presented here, form the literature basis used by this work to identify the essential issues related to mobile context-aware computing.

The discussion is partitioned according to the primary aims of the research efforts [23], which are *Conceptual Frameworks*, *Service Platforms*, *Appliance Environments* and *Computing Environments*. Section 3.1 discusses two projects which aim at presenting a *Conceptual Framework* for building context-aware applications. Section 3.2 explores two projects committed to building context-aware *Services Platforms*. Section 3.3 discusses the projects whose initiatives are in the field of *Appliance Environments*. Section 3.4 explores projects in the area of *Computing Environments*. Finally, Section 3.5 gives a comparative table showing the differences between the identified initiatives in terms of a set of context-awareness issues.

3.1 *Conceptual Frameworks*

Conceptual frameworks for context-aware systems define a conceptual basis to support the development of context-aware and adaptive systems and applications [23]. They aim at facilitating the gathering of information from sources such as sensors and context providers; performing interpretation of data; carrying out dissemination of contextual information to interested parties in a scalable and timely fashion; and providing models for programming context-aware applications. Examples of context models are the context toolkit [9], a project by the University of Berkley, and the cooltown [26], a project by Hewlett-Packard.

3.1.1 *Cooltown Project*

The project proposes a web-based model for context [26]. Within this model, physical entities are divided into three categories: people, places and things. Examples of places are homes, offices, shopping malls, etc. Examples of things

are printers, radios and automobiles. People are the users of things and the occupants or visitors of places. Places have a special role as the venue or container for people and things.

The goal of the project is to expand the view of physical entities to a virtual world of web content, in which people, places and things are web-present. Things become web-present by embedding web-servers in them or by hosting their web-presence within a web server. Places become web present by organizing web things into collections under the management of a web service called PlaceManager. People become web present by offering global web home pages with WebLink services to facilitate communications between individuals and by offering information via location-specific PlaceManagers.

Points of web presence of people, places and things are obtained through discovery systems and by sensing URLs and identifiers. There are two approaches to discovering the URLs of entities via a sensor: either to sense that URL directly or to sense an identifier from the entity, which is then looked up to obtain the URL.

A place is a context for service provision, based on an underlying physical domain. For instance, a café covered by Bluetooth or a railway station covered by WaveLAN connectivity. Web present places are hyperlinked collections of the web presences of people, places and things, which create a physical and contextual organization of place information. The PlaceManager is a service that stores those hyperlinked collections. It is responsible for organizing the set of resources present in a place and the services around them. The information that the PlaceManager provides depends on the clients' permissions and client device's functional capabilities.

Location awareness is based around the concept of a place. Beacons transmit URLs corresponding to an entity (things, people or places), enabling devices near the beacons to discover and access their local places. PlaceManagers virtually organize the contents of a place, which are accessed through portals. Portals are the gateways to space's services. A PlaceManager keeps track of the current devices within a place and generates dynamic web pages that reflect their current context.

The Cooltown framework clearly has limitations. It does not allow defining a common view of context; instead, it allows arbitrary web description of context. Interpretation and subscription of context events are outside the scope of the framework.

3.1.2 *Context Toolkit*

The context toolkit conceptual framework focuses on programming with context rather than on context representation [9]. The toolkit was designed to tackle the following requirements for dealing with context:

- Separation of concerns: separation of acquiring context and handling context;
- Context interpretation: extend notification and querying mechanisms to allow applications to uniformly retrieve context from distributed systems;

- Transparent, distributed communications: Application and sensor designers do not need to bother with communication protocols and with the design and implementation of encoding schemas for passing context information;
- Constant availability of context acquisition: context-aware applications should not instantiate individual components that provide sensor data, but must be able to access the ones that are available. Furthermore, multiple applications may need to access the same piece of context. Therefore, components that acquire context must be executing independently from the applications that use them;
- Context storage and History: constantly availability of context to maintain historical information;
- Resource discovery: discovery of context providers (or rather its software interfaces). For that, applications need to know what kind of information a provider offers, where it is located and how to communicate with it (protocol, language and mechanisms to use).

Context widgets, interpreters, aggregators, services and discoverers are the categories of components in the conceptual framework, which are used to address the aforementioned requirements. Their functionality is explained in the next paragraphs.

Context Widgets are software components that provide applications with access to context information from their operating environment; they hide the complexity of the actual sensors used from the application; they abstract context information to suit the expected needs of applications; they provide reusable and customizable building blocks of context sensing. From the application's point of view, the widgets encapsulate context information and provide uniform operations to access context.

Interpreters are for raising the level of abstraction of a given context. For example, location may be expressed in the form of latitude and longitude or at higher levels such as city or street. It is also possible to infer new pieces of contextual information from more than one context source.

Aggregators collect multiple pieces of logically related context information into a repository making it available within a single software component.

Services are provided by components that execute actions on behalf of the application. When a certain combination of conditions is met, an action should be taken by the application.

Discoverers are responsible for maintaining a registry of what capabilities exist in the framework. Therefore, it keeps track of the available widgets, interpreters, aggregators and services. When any of these components is started, it notifies a discoverer of its presence and capabilities and how it can be contacted.

3.2 *Service Platforms*

Service platforms provide an infrastructure, which aims at the rapid creation and deployment of context-aware services, while also offering dynamic service

discovery, dynamic deployment of adaptive applications addressing issues of scalability, security and privacy. Examples of service platforms are the M3 architecture [25] from the University of Queensland, Australia and the platform for adaptive applications [14] from the Lancaster University, U.K. We will elaborate on them in the next sections.

3.2.1 *Platform for Adaptive Applications*

The main goal of this research is to build a unified architecture that support multiple contextual attributes coupled with a user driven adaptation control mechanism. It is believed that this approach is able of overcoming the limitations of current approaches for supporting adaptive mobile applications, which are lack of coordinated adaptation and lack of dynamic adaptation mechanisms.

Different adaptation policies, which coexist in an environment scenario in which the system resources are shared among multiple applications, can result in conflicting problems. Moreover, this scenario also requires global adaptation mechanisms in the system level, not only in the application level, in order to provide system's consistency as whole, given the current context and the user requirements.

The following system architectural requirements are addressed by this effort:

- Support to a common space for handling an extensible set of adaptation attributes (new contextual attributes for triggering adaptation can be added);
- Support the control of adaptive behavior across all components involved in the interaction. One of the main problems of current approached is that the applications themselves are responsible for triggering adaptive mechanisms when the underling infrastructure notifies them about the changes. In order to support flexible and coordinated adaptation it is necessary to have triggering of adaptation on a system-wide level. The ability of deciding when and how an application should adapt is pushed into an external entity, with cross-application knowledge, while the adaptive behavior is still a part of the application's characteristics;
- Support the notion of system-wide adaptation policies. Such policies should enable a mobile system to operate differently given the current context and the requirements of the user.

The main characteristics of the architecture proposed by this project are (i) discovery and control of services offering contextual information and (ii) coordination of adaptive behavior of the system based on changes in context. In summary, the platform discovers available context information in the system's environment and manipulates the contextual information in a context database. Context-aware applications expose their adaptive mechanism to the platform by registering with the application database. The adaptation control driven by adaptation policies (as specified by the user) coordinates the coexisting applications according to changes in the context.

3.2.2 *M3 Architecture*

M3 is a proposal of an open architecture for ubiquitous systems. It aims at defining an open component-based architecture, which is able to describe complex computational context and handle different types of adaptation for a variety of new and existing ubiquitous enterprise applications [25]. For that, the architecture uses a component based modeling paradigm and an event-based mechanism, which is able to give flexibility in terms of dynamic reconfiguration and adaptation.

The following requirements were taken into account when designing the architecture:

- **Reactivity:** the architecture should be easily programmed to react to context changes;
- **Open dynamic composition:** the architecture should be open to allow interoperation with other component, such as legacy applications or new components. Dynamic Composition (plug and play) configuration and customization of services is essential in a ubiquitous environment.
- **Enterprise focus:** the architecture should be able to understand the existing roles, and the relationship among them, in an enterprise environment. For that, it is necessary to have abstractions that capture the purpose, scope and policies of the system in terms of behavior expected of the system by other entities within the enterprise.

The main characteristic of this architecture is the possibility of programming the platform based on a coordination specification language, called MEADL. MEADL specifies the coordination of events. Its specifications provide means to dynamically configure interactions between components, to change dynamically the communication paradigm and to provide session management.

Events belong to roles; therefore MEADL specifies the coordination of roles. Users, ERP (Enterprise Resource Planning) servers, network protocols or dedicated managers can fulfill roles. A role's description is the interface of tasks that a component can provide. A role's duty is described in a set of input/output events.

3.3 *Appliance Environments*

The goal of appliance environments is to support interoperability among collections of appliances [23]. Examples of such environments are the Ektara [8] environment, project by MIT and the Universal Information Appliance [15], project by IBM.

3.3.1 *A Universal Information Appliance (UIA)*

The goal of this project is to make a universal user interface deployed in a PDA-like device or a wearable computer, to interact with any application, access any information store, or remotely operate any electronic device [15].

The challenges overcome by this infrastructure are:

-
- *Device Requirements*: Users will want to use a range of different devices. One solution to cope with device diversity is to define a common set of elements the devices need to have in order to be a UIA. These elements are *output mechanism* (visual, tactical or audio), *input mechanism* (touch screen, speech recognition or visual sensor), *local data storage* and *network communication* that will relay data and event messages among devices and among devices and services;
 - *Communication Infrastructure*: It is necessary to define an infrastructure in which the UIA can communicate. This infrastructure should include mechanisms of connecting the UIA to the various information, interface and application servers. Moreover, this infrastructure should include some kind of discovery and retrieving of services.
 - *Seamless Integration*: Integration of the UIA in user's daily lives.

In order to tackle the device requirements, a platform independent application and interface language and local storage system were developed. These were implemented on a standard PDA (the IBM WorkPad). For the wireless connection, the existing standard messaging system was used. For the communication middleware, TSpaces was used. TSpaces is the IBM middleware capable of gluing system components together by combining data management, computation and communication.

MoDAL is a platform independent language, which is used to describe remote interfaces. MoDAL applications are uploaded dynamically into a user's device from their corresponding services, and are tailored to the user's device and preferences. The benefits of this communication model are that it offers distribution transparency, can support a range of interaction types, including event and stream interactions, and removes the need for resource discovery.

3.3.2 Ektara Architecture

The Ektara architecture [8] is a distributed computing architecture for building context-aware ubiquitous and wearable computing (UWC).

The requirements of the system are:

- Centralized management of competing demands for the user's attention;
- Decentralized contextual resource discovery and allocation;
- A uniform, decentralized mechanism for contextual information storage and retrieval;
- Flexible context sensing and classification based on heterogeneous sensors;
- Strong cryptography for authentication and privacy;
- Open standards for the seamless integration of wearable and ubiquitous computing resources.

The design philosophy of the architecture puts the user needs in the center of the design process.

The components of the architecture are:

- Context-aware interaction manager (CAIM): provides a uniform framework for interaction between applications and user trying always to minimize the need for explicit user input while maximizing the relevance of the information provided. The CAIM's model becomes part of the user's personal profile and it is located in the wearable device being available wherever the user goes.
- Dynamic decentralized resource discovery: this framework allows applications and services to find and use resources that match semantic descriptions of functionality and context. To make that possible, the wearable application components need to be registered in a registration service with their semantic descriptions, capabilities and any additional contextual information it chooses to provide;
- Contextual information service (CIS): it is a distributed database service, which provides the wearable applications and services a uniform means of storing and retrieving contextual information.
- Perceptual context engine (PCE): it is capable turning sensor data and other sources of information into symbolic context descriptions;
- Strong cryptographic security and authentication: communication between CIS servers and clients is typically executed in non-trusted hosts and across non-trusted networks. Therefore, security and authentication in the wearable environment must be achieved with a decentralized public-key cryptography infrastructure.

3.4 *Computing Environments*

Computing environments embrace the systems that broadly address the ubiquitous computing goal of providing context-aware computing (anytime, anywhere) by decoupling users from devices and viewing applications as entities that perform tasks on behalf of the user [2]. There are several projects in this classification, such as PIMA [2], project by IBM and Portolano [13], project by University of Washington.

3.4.1 *Portolano*

The Portolano project defines an infrastructure for ubiquitous computing addressing issues related to the areas of user interfaces, distributed services, and networking infrastructure.

The main characteristics of this infrastructure include:

User Interfaces

It is believed that new models of users interactions have to be developed. User movement, proximity of devices, and embodied information presentation will

augment the keyboard, pen, audio and video interface used nowadays. The focus of the user interaction must shift to user intent and expectation and away from the execution of explicit commands. Intelligent interpretation of data gathered from different sensors, identification of tags, and on-line services will replace many user explicit directives of today. The challenge is to make the UI fit so well into the environment that it becomes invisible inasmuch as the user is aware she is interacting with a computing device.

Distributed Services

The emphasis must shift from abstract capabilities and specific infrastructure to applications in which users can easily relate. Different user interfaces, appropriate to their contexts, should be able to interact with the same service.

Networking Infrastructure

The network should provide robust data transfer with replication and discovery. User must rely on their data arriving where it needs to go without their explicit intervention. Therefore, the network must be data centric; transmission, routing, authentication and resource reservation should be handled independently of the location of the user who inserted the data.

3.4.2 PIMA

Unlike the Portolano project, whose emphasis is on the infrastructure, the PIMA project proposes an application model for ubiquitous computing. The main idea is that application logic must be decoupled from details that are specific to the run-time environment. To achieve this vision, a new application model is proposed considering the life-cycle of a ubiquitous application. This life-cycle is divided in parts: design-time, load-time and run-time.

Design-Time

The time when the developer creates, maintains and enhances the application. Applications should not be designed to a specific type of device. Moreover, the user interface of the application must not include any information specific to a device or set of services. Instead, the application front-end should be device-neutral. Developer should not make assumptions about services as well. Services should be specified in an abstract manner. Therefore, not known services at design time will still be available at run-time.

Load-Time

According to the project envision, the traditional load-time approach is not supported. Devices must dynamically discover what applications are available, and the system must adapt the applications to the device resources available. An application must be specific in terms of its requirements, the device must be described in terms of its capabilities and some intelligence must be used to match the applications' requirements and device capabilities. For those reasons, the system must be dynamic at load-time. That is, the tasks that a user wishes to perform may depend on the physical surrounding. Such tasks are enabled by contextual services. The systems must be able to discover and compose those services in order to perform specific tasks.

Run-Time

Resources must be constantly monitored in a run-time environment, so that application can appropriately adapt to those resources. In addition, the run-time must respond to changes initiated by the user. Moreover, the run-time must allow a user to initiate and constantly perform a task, despite of changes in the environment. Hand-off of tasks from one environment to another should be supported and finally, the run-time must be able to take advantage of services provided by the environment.

3.5 *Discussion*

Table 1 shows a comparison between the initiatives addressing some of the issues related to building a ubiquitous infrastructure, in special with respect to context-awareness issues. The following topics are addressed in the comparison:

- Support for device heterogeneity: it is expected that device heterogeneity in computer systems will no disappear in the future, but instead will increase as the range of computing devices widens. Heterogeneous devices will be required to interact seamlessly, despite differences of hardware and software. This implies that the infrastructure has to provide solutions that enable arbitrary device interactions;
- Support for device mobility: device mobility is a basic requirement to allow user mobility, which is the pillar for mobile context-aware computing. Nevertheless, device mobility introduces problems such as the maintenance of connections as devices move between areas of differing network connectivity, and the handling of network disconnections. Therefore, it is the role of the computing infrastructure to cooperate with applications in order to perform tasks related to device mobility;
- Management of application mobility and distribution: mobility and distribution of software is a reality when dealing with mobile context-aware computing. The supporting infrastructure should be able to provide transparency of distributed communications addressing issues like mobility, synchronization and coordination;
- Support for context-aware issues: handling context introduces problems, especially when applied to distributed systems such as context storage, context gathering, context management, context interpretation, etc. The computing infrastructure should give support to the fundamental issues related to context-awareness;
- Support for adaptation: context-aware computing relies on the adaptation (reaction) of users and devices in response to context changes. It is the role of the infrastructure to facilitate adaptation;
- Support for rapid development/deployment of applications: ideally, the computing infrastructure should provide services, such as adaptation, context gathering and management, resource discovery, transparency, so that, application developers do not need to concern with these fundamental issues;

- Management of user context: in order to provide added-value services to users based on their context and their preferences, the infrastructure should maintain context data related to users, including their capabilities, preferences and current context (activity, location, time, etc);
- Support for user mobility: user mobility between devices should become transparent. Software components should migrate transparently from one device to another. The task of finding out if the migration is needed when actually performing the migration, should ideally be performed by the computing infrastructure.

Issue	Conc. Frameworks	Service Platforms	Appliance Environments	Computing Environments
Support for device heterogeneity			•	
Support for device mobility			•	
Management of application mobility and distribution				
Support for context-aware issues	•	•		
Support for adaptation		•		•
Support for rapid development / deployment of applications		•		•
Management of user context	•			
Support for user mobility				

Table 1 – Comparison between the initiatives

From the comparative table, we can conclude that some of the issues are present in those research projects but a considerable amount of challenges have so far been little addressed. It is still challenging to develop an infrastructure that integrates solutions to all the presented issues.

4 *The WASP Platform Requirements*

The WASP platform should provide generic functionality to address the basic aspects and challenges of context-aware computing concerning a services platform. The goal of this chapter is to identify, define and refine the essential requirements that cover these aspects and challenges.

This chapter is structured as follows: Section 4.1 discusses the requirements related to the manipulation of *Contextual Information*. Section 4.2 elaborates on the requirements for the interactions between the platform and its environment. Section 4.3 presents some *General Requirements*. Finally, Section 4.4 discusses some final remarks.

4.1 *Contextual Information*

The WASP platform should be capable of gathering contextual information from different sources and adapt to them according to the user needs and system capabilities. For that, common understanding of contextual information is required. *Context Representation/Modeling* and *Context Storage* are the main requirements related to contextual information.

4.1.1 *Context representation/modeling*

As discussed in Section 2.3, there are different kinds of context with different characteristics and complex interrelationships. The WASP platform should define and use systematic approaches or context models that capture the nature of contextual information.

Furthermore, interoperability and knowledge sharing in a distributed system becomes challenging when the systems are based on ad hoc models of context. The determination of the context model should consider interoperability issues, since the platform is expected to operate in a distributed environment.

The WASP platform context model should be extendable to allow the deployment of new kinds of contextual information that have not been anticipated during the

platform design. For instance, the determination of the model could consider, initially, the categories *location, users, objects and devices* (Section 2.3.2). These categories could be extended to future deployments of other types of context, for example, *time* and *user activity*.

For the purpose of this thesis, we focus on the definition of a simple extensible context model. More sophisticated context modeling approaches, for instance, using ontology-based techniques, are to be considered elsewhere in the WASP project.

4.1.2 Context storage/retrieval

Contextual information will be made available through the services platform. In order to keep track of the information, the platform should gather context from different sensors (or context providers), process it, and store the results. Some applications may require that this information to be maintained and preserved over time. For instance, an application could request the location of a person, in a given date in the past. Furthermore, keeping history of contextual information is particularly interesting to allow context inference based on past occurrences. For example, the inference of the speed of a user from the latitude and longitude changes over time.

Different representations can be used for communication, processing and storage, each optimized for its own purpose taking into account the software and hardware that are used for this. It is important that the interpretation of these representations is unique, and that the transformations are supported, i.e., the parties involved in the communication must agree upon the semantics of the information.

4.2 Platform Interactions

The platform interacts with three systems: WASP Applications, Services Providers and Context Providers, as depicted in Figure 4.

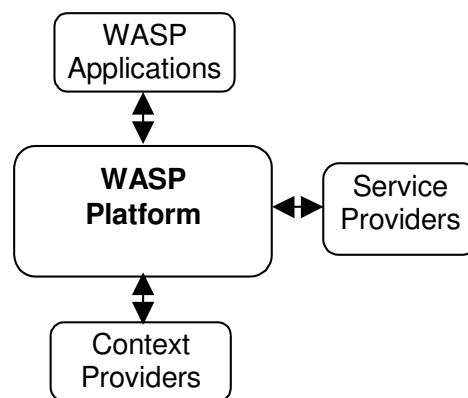


Figure 4 - Platform interactions

The next paragraphs discuss those interactions and the challenges related to them:

- Interaction Platform-Context Providers: support for different kinds of context providers (sensors or third party context providers);

- Interaction Platform-Application: reactive behavior and coordination among different applications;
- Interaction Platform-Services providers: discovery and publishing of services.

4.2.1 Support for different kinds of context providers

The platform should be open to new kinds of third party context providers and to new kinds of sensing mechanisms, not only 3G Networks. Context providers supply information using different communication protocols and in semantically different formats. Therefore, components that hide the process of acquiring context from sensors and providers are necessary.

For this reason, the design of the platform should take into account an adaptation layer that makes contextual information from different providers uniformly presented (well understood) to the rest of the platform. This component should be able to hide the details of the provider (sensor or third party) and how to acquire context from this process completely transparent to the other components in the platform.

An example that gives an idea of a solution for this challenge can be found in the Context Toolkit conceptual framework [9] (Section 3.1.2). Figure 5, shows an example configuration of the context toolkit for two different sensors. Each of the sensors provides contextual data to a widget, which encapsulates the sensor storing the data and also looking for higher-level abstraction of contextual data.

For instance, a widget could be responsible for translating latitude and longitude to a city and a street. An aggregator is able to collect context from the widgets that provide relevant context for a given task. For example, an aggregator could collect the location of several different users and check if they are crowded in the same location in order to verify the possibility of a traffic jam.

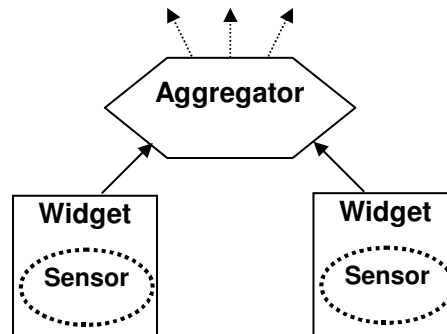


Figure 5 - Example configuration of the context toolkit conceptual framework

In summary, the desired platform support for context manipulation should address issues such as context gathering (the platform should be able of collecting context from *Context Providers* and/or *sensors*), context aggregation (the platform should be able of logically aggregating multiple pieces of related contextual information), context inference (the platform should be able of inferring context from other context(s)).

4.2.2 Reactive behavior

Applications should be able to respond to their dynamic environment. Therefore, the platform has to support the applications in this process since the platform is the one aware of the changes in the user environment. But how to support a large and growing number of different context-aware applications without having to upgrade the platform each time a new application is deployed? One possible solution is to give some intelligence to the platform by exposing reaction mechanisms, i.e., applications have to “teach” the platform how to react to certain correlations of events [20]. Event correlation is the designation of relationships between multiple events. Those relationships can be logical operators, such as AND, OR and NOT. Suppose E_1, E_2, \dots, E_n are known events to the platform. A possible correlation of these events is $((E_1 \text{ OR } E_2) \text{ AND } E_3) \text{ OR NOT } E_6$. On one hand, if the result of the formula turns to be true, it enables some kind of action to be triggered. On the other hand, if the result turns to be false, it disables the action to be triggered.

Figure 6 depicts the desired architectural models of interaction to be supported by the WASP platform, abstracting from the 3G networks. The upper layer represents the application layer and the lower, the supporting platform. The request/response model (passive platform behavior) is the one where the reactive behavior of the platform is just to respond to the applications requests. In the event-driven model (event-driven platform behavior), applications expose to the platform the desired reactive behavior by means of a subscription. The subscription is based on correlation of events and programming of actions. The models are also depicted in the sequence diagrams presented in Figure 7 and Figure 8.

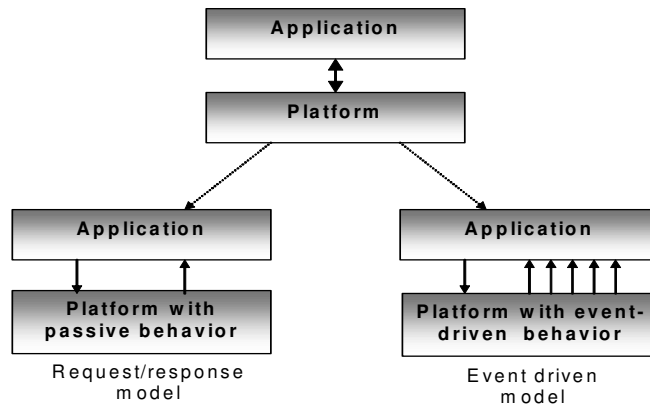


Figure 6 - Different models of interaction between application and platform

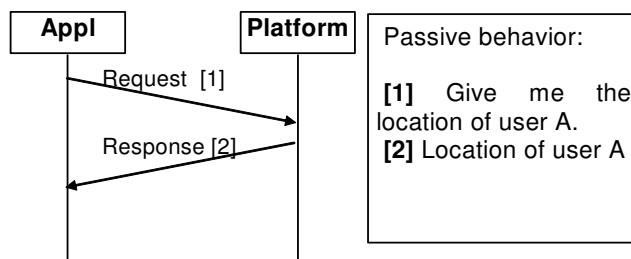


Figure 7 - Example scenario for platforms with passive behavior

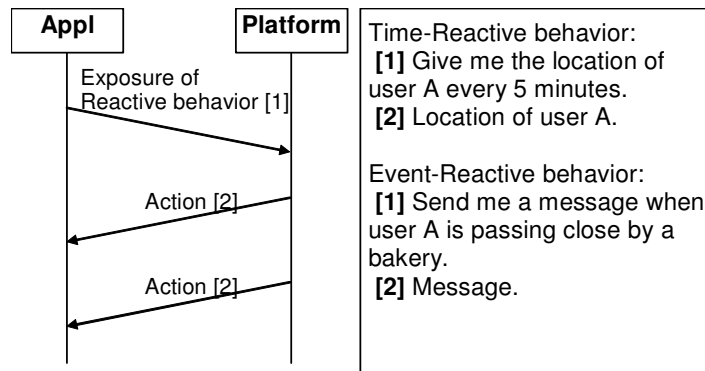


Figure 8 - Example scenario for platforms with event-reactive or time-reactive behavior

For example, consider the simple user’s scenario “*Send me a reminder of buying bread when passing close by a bakery*”. In this scenario, the user is manipulating a reminder application, which is supported by the platform. In order to make of use of the platform, the application needs to express that user *John* should receive a *reminder of buying bread* when passing less than a 100 meters from a bakery. In addition, this request should remain in the platform because *John* wants to be reminded every day. The following expression is an abstract view of how the platform could be programmed by an application.

ACTION sendMessage
GUARD closeBy
 (entity:user:ID:location, entity:bakery:any:location)

The platform understands the *user* and the *bakery* as entities. *Close by* is a function that returns *true* if the physical location (context) of both entities are close, for instance, less than 100 meters, and *false* otherwise. The combination of this function and the user context *location* forms the logical condition for the action to be triggered. Finally, “*Send a reminder*” is the action that should be triggered every time the condition turns *true*.

The platform has to keep track of all possible entities (*user, bakery, museum, restaurant, etc.*) involved in an event correlation as well as their *context*. In the aforementioned case, based on the *user location*, the platform knows how to find the bakeries in the user’s surroundings.

Ideally, the platform should support a large (and potentially growing) range of context-aware applications. Therefore, it is necessary to embed flexibility in the platform in order to make it suitable for most applications. Furthermore, application-platform interactions should be dynamically configured allowing application deployment during the platform run-time. To address this issue, we need a solution that allows applications to expose their needs to the platform during run-time. A possible way to address this requirement is to permit applications to subscribe to the platform by means of adding *application subscriptions* to the platform. *Application subscriptions* would be written in an established (agreed) language and the main elements of an application subscriptions would be a trigger (*action*) and a conditional expression (correlation of events involving *entities* and their *context*). The basic semantics of this language would state that when a conditional expression turns *true*, the *action* should be triggered.

The UML diagram depicted in Figure 9 shows a possible representation of how the event trigger, the conditional expressions and actions would relate: The platform supports many applications and each of them would have several subscriptions. Each subscription would contain at least one trigger and when the conditional expression is satisfied, the action should be triggered.

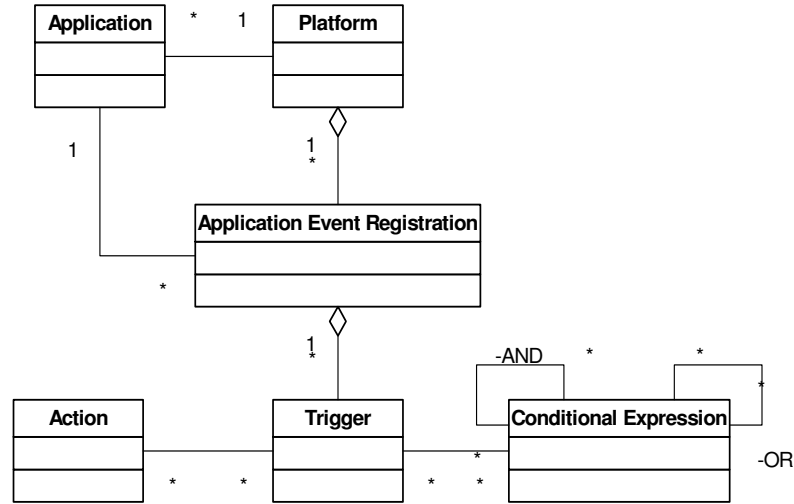


Figure 9 - Representation of reactive expression

4.2.3 Coordination among different applications

Different reaction mechanisms programmed by different application can give rise to conflicting problems when providing a service for the same user. For instance, take an example in which the user uses two different applications in his device, a reminder and a sleep mode application. The sleep mode application turns the device to a sleep mode in a certain situation, for example inside the movies, in a meeting or when he/she is driving. The reminder sends reminder messages when a correlation of events, determined by the user, happens.

A component with cross-knowledge capabilities, i.e., a component with knowledge of different sources in the architecture, for example a monitoring component [22], can be responsible for managing the coordination of events by checking the user context and preferences. This way, side effects generated by conflicting reaction mechanisms can be avoided.

The sequence diagram depicted in Figure 10 and Table 2 illustrate this scenario.

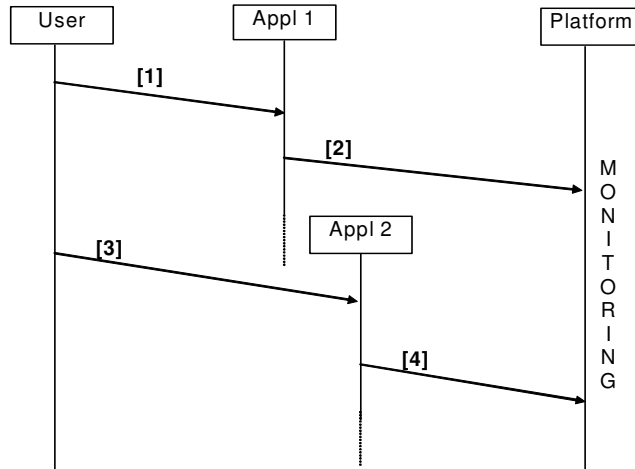


Figure 10 - Bakery scenario

Message Number	Message Contents
[1]	"Remind me of buying bread when passing by the bakery"
[2]	ACTION sendMessage (entity:user:ID); GUARD (entity:user:ID:location closeBy entity:bakery:all:location)
[3]	"Sleep mode when I'm driving"
[4]	ACTION action:sleepMode (entity:user:ID); GUARD entity:user:ID:driving
[Monitoring]	The scenario "the user is driving and passing by the bakery" is a conflicting situation. The platform could monitor this situation by checking the user preferences. By doing that, the monitoring component would realize that the user priority is to not be disturbed when driving (he/she has concentration problems) even when passing by the bakery.

Table 2- Bakery scenario

4.2.4 Discovery and publishing of services

Service provision by third party service providers is the essence of the service platform. Discovery and publishing of services can be done by internal elements or/and by external elements, being opened and shared with others application environments/platforms.

Entities like museums, restaurants, supermarkets, bakeries, schools and hospitals want to expose their services to the platform, so that, depending on the user necessity, taking into account his/her context (or correlation of), those services can be used. In the aforementioned reminder application example, the platform uses information about bakeries, in this case, their location. There are cases in which the platform should be programmed to directly use the services of service providers. For example, the platform could be programmed to immediately order bread when the user is approaching the bakery. Therefore, the user does not need

to be bothered with selecting and paying activities; this could be automatically done by the platform.

The platform should provide a mechanism to support discovery and publishing of services. A centralized service discovery approach, which has been largely explored in recent works, is Universal Discovery, Description and Integration (UDDI) [49]. It provides a directory service where service providers and service requestors come together to satisfy their needs.

Related work inside the WASP project intend to add functionality to the UDDI in order to improve its capabilities [37]. It is claimed that UDDI lacks on semantic description, process specification and ontology support. Their aim is to implement an enhanced UDDI server, capable of storing, matching and retrieving semantically rich service profiles that contain contextual information.

4.3 *General Requirements*

4.3.1 *Support for security and privacy services*

The platform should be able to gather important and perhaps private information from different parties. Therefore, security and privacy services are clearly a necessity in a context-aware environment. Which information can be published? To whom? Can it be available forever or only for a defined period of time? These are example questions that should be addressed when dealing with privacy aspects of the platform.

Although it is a very important issue in context-aware computing, privacy has not been properly enforced in available platforms. There has been very little attention to this concern. Examples of relevant work can be found in [28, 29], which define (i) guidelines on how to design a system based on a set of fair information practices common in most privacy legislation and (ii) a privacy architecture that is part of a compulsory security framework that allows the use of appropriate security policies and authorization services.

The WASP project approach to address privacy concerns should be to study (and prove the suitability of) the current developments in context-aware privacy control, proposing improvements to existing solutions and/or developing of new ones. For example, the WASP project could investigate the Privacy Preferences Project protocol (P3P), which is a protocol primarily intended to describe privacy policies of web sites. In order to translate P3P to web-services domain, enhancements on P3P need to be proposed. Privacy and security concerns are to be considered elsewhere in the WASP project.

4.3.2 *Charging*

There are important issues related to charging in a commercial service platform such as: which parties are going to charge and for which services. A possible model for the WASP platform could describe that users pay for the services provided by applications. Similarly, applications pay the platform for services and the platform pays for the usage of 3G Networks. This example charging configuration (with three main levels of charging) is depicted in Figure 11.

Once a basic charging model is chosen, a business model for the WASP platform should be defined. This business model would define requirements for the assignment of business responsibilities among the different parties involved in the WASP project. An example of information defined by this business model could be how to measure (and charge for) the service: per access, per information quantity or per time.

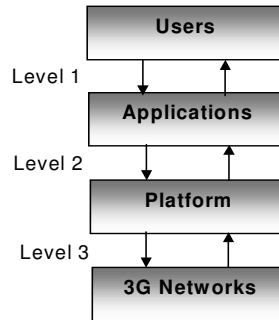


Figure 11 - Charging levels

4.4 Concluding Remarks

In this chapter, we have discussed some of the essential requirements to be satisfied by a context-aware services platform. Having elaborated on the essential requirements and having knowledge on the existing solutions that overcome some of the identified challenges, we are able to propose an architecture for the WASP Platform. The design of the proposed architecture is reported on Chapter 5.

Issues of scalability, performance and the use of standards should also be addressed as requirements for the WASP platform. Scalability concerns, in particular, need to be considered since the platform will provide services to a large number of applications, users and service providers. Therefore, the platform should be able to manage a large volume of context information, user profiles and preferences and it also has to keep track of a large number of event-correlated rules, which means manipulation of data from different sources in the platform. The platform should be scalable in order to cope with this heavy and potentially growing load. Therefore, architectural decisions such as distribution of components and separation of context sensing (and filtering) and context processing, should be taken into account.

Furthermore, devices and applications from different vendors should interoperate seamlessly with the platform. Therefore, interoperability issues should be taken into account. One way to cope with diversity is the use of standards like CC/PP, P3P and Web Services, which are focus of investigation in the WASP project.

In this thesis, we address a non-trivial subset of the identified requirements during the design phase. In particular, the following challenges are directly explored by the design:

- Support for gathering, interpreting and storing of contextual information;
- Reactive behavior. This challenge implicitly addresses dynamic application deployment during the platform run-time;

-
- Coordination among different applications;
 - Support for dynamic service discovery.

The following challenges are not explored by the design:

- Context representation/modeling;
- Security and privacy;
- Scalability and performance;
- Charging.

One of the most challenging requirements we identified is the *Dynamic application deployment during the platform run-time*, which is associated to the application-platform interaction. The architectures we have investigated do not address this challenge with the level of configurability we require. As can be seen in Chapter 5, we spend 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. Embedding this level of flexibility into the platform facilitates the creation and the dynamic deployment of a large range of context-aware applications.

5 *The WASP Platform Architecture Design*

This chapter reports on the design of the WASP platform architecture. Based on the requirements discussed in Chapter 4, we have identified the components of the architecture and the interactions between these components.

This chapter is structured as follows: Section 5.1 gives an overview of the architecture, mentioning the main components and their main function. Section 5.2 presents the WASP platform context model. Section 5.3 details the *Context Interpreter* component, which essentially allows the encapsulation of *Context Providers* making contextual information uniformly available to the rest of the platform. Section 5.4 explores the *Repository* components, which contains essential information necessary to manage the application subscriptions. Section 5.5 discusses the heart of the platform, the *Monitor* component. This component is responsible for managing the application subscriptions. This section also presents the WASP Subscription Language (WSL), which is the language to represent subscriptions.

5.1 *Overview of the Architecture*

Figure 12 depicts a high level view of the platform architecture together with the platform interaction systems. Figure 13 shows a refined view of the components.

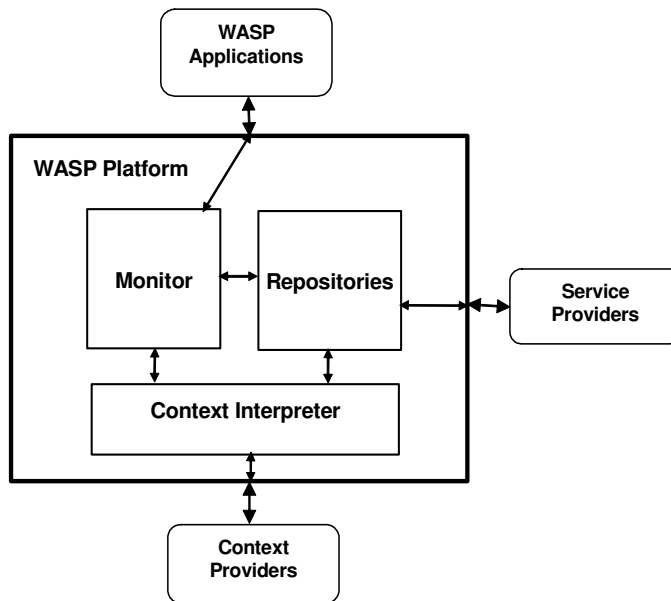


Figure 12 - WASP platform architecture

As aforementioned, the platform forms the system environment for context-aware applications (*WASP applications*). *Services Providers* are business parties that foresee opportunities to profit from offering their services through the platform. *Context Providers* are the parties responsible for providing contextual information.

The WASP platform architecture is composed by three main elements: *Monitor*, *Repositories* and *Context Interpreter*. The *Context Interpreter* gathers contextual information from *Context Providers*, manipulates context and makes it uniformly available to the rest of the platform. The *Repositories* are responsible for supporting the *Monitor* with knowledge of the elements involved in the platform. For that, some of the elements in the *Repositories* module collect information from the *Context Interpreter* and others use the services of *Service Providers*.

The *Monitor* is the core of the platform since it is the module responsible for interacting with *WASP Applications*. Therefore, it is responsible for interpreting and managing the application subscriptions. In order to perform its operations, it gathers information from the *Repositories* module and from the *Context Interpreter*.

In our architecture design, we give emphasis to the application-platform interactions issues that deal with the dynamic deployment of applications. Therefore, the design of the *Monitor* module is an outstanding part of this Chapter. In particular, we have developed a language, called the WASP Subscription Language (WSL), which allows application-platform interactions to be dynamically configured. The use of this language greatly leverages the level of the platform configurability. We will elaborate the abovementioned architectural elements and the WSL in the next sections of this Chapter.

The UML diagram depicted in Figure 14 describes a possible configuration of the context model proposed for the WASP platform, called the *Entity Model*. The model is not complete, since it shows only an example configuration with entities *restaurant* and *user* and their relationship with context *location*. Along the platform usage, this model can 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, 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 run-time. The lower levels of instantiations are called the *model* level and the *object* level. These levels are dynamically changeable during the platform run-time. They represent instances of the *Metamodel* and the *Model* levels, respectively.

In the first level of instantiation (*Metamodel* level) it is defined that *Entity Types* are associated with *Context Types* and *AccessModelType* is an attribute of this association. The association class *AccessModelType* provides information over how to gather the context from the *Context Interpreter* using the attribute *AccessModel*. This attribute can be *Request/Response* model, *Time-Driven* model or *Event-Driven* model. These models are discussed in Section 5.3.1.

Figure 14 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*).

Once our *context model* is defined, it must be taken as a reference by the systems that interact with the platform in order keep the common understanding of the information semantics.

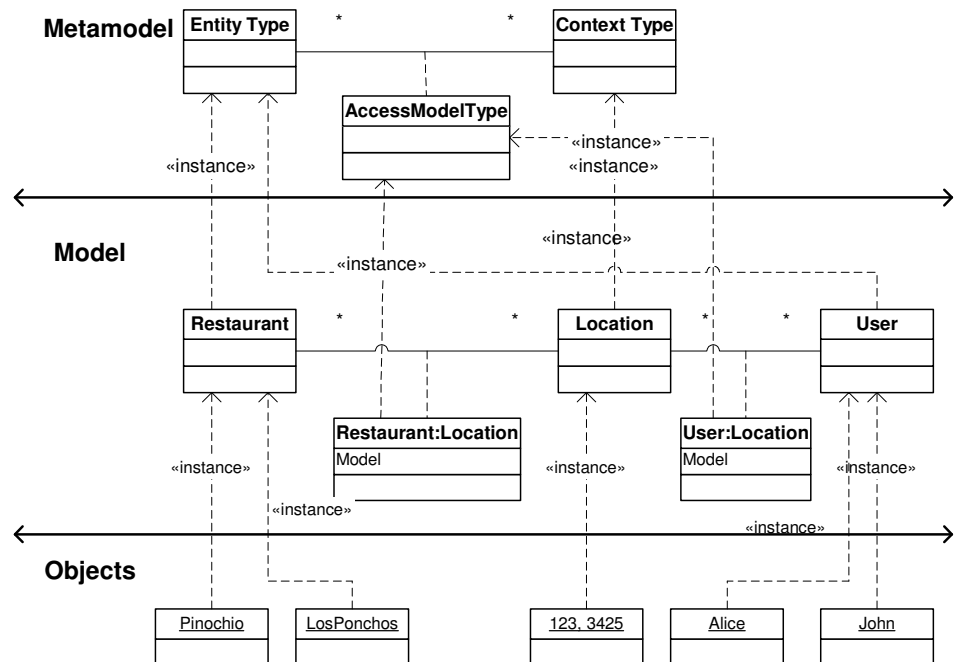


Figure 14- Platform instantiation levels

A combination of entity type and context type is associated with an access model. For instance, the combination **user.location** could be associated with any of the first three models. With the *Request/Response* model, the location of the user is provided upon a request. With the *Time-Driven* model, the location of the user is provided from time to time. With the *Event-Driven* model, the location of the user is only provided after the location value has been updated. The entity type **restaurant**, in combination with context **location**, would be associated with the *Request/Response* model since the location of a restaurant is unchangeable.

Instances of entity type *user* are stored in the *User Profile* repository. Instances of *entity types* that are service providers (*restaurants, museums, etc.*) are stored in the *Service Discovery* component. Instances of *entity types* that are not *users* and not service providers (*road, building, room, etc.*) are stored in the *Entity Repository*.

It would be possible to define (hierarchical) relations between *entity types*. This could vary from simple categorizations of entity types to complex *ontologies* [6]. A common representation of this knowledge is essential for the interoperability of the platform and its environment. *Ontologies* are believed to be a promising modeling technique for building context-aware systems, as discussed in Section 2.3.3. Therefore, the WASP context model can potentially profit from the use of *ontologies*. *Ontology-based* techniques to model contextual information are currently being investigated by the WASP project.

5.3 Context Interpreter

The *Context Interpreter* gathers contextual information from *Context Providers* (sensors or third parties context providers) which may use different communication protocols and semantically different contextual representation, making contextual information uniformly available to the platform. Therefore, it is responsible for tackling the *support for different kinds of context providers* requirement, mentioned in Chapter 4, Section 4.2.1.

The *Context Interpreter* is able of gathering and providing context in several levels. This is illustrated in Figure 15. The rectangles represent the layers in which contextual information is gathered/ provided.

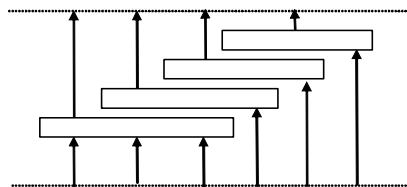


Figure 15 – Layers of contextual information

Figure 15 does not show this explicitly, but the upper layers can access all the layers below it, not just the one immediately below. In fact, the *Context Interpreter* is able of taking raw context from the context provider, semantically interpreting it and then, based on this interpretation, inferring new contexts. This process can be performed as many times as necessary.

For example, suppose the latitude and longitude of a user is being pulled from a *Context Provider*. The *Context Interpreter* is able to assess the speed of a user

from the latitude and longitude changes over time. Moreover, it may be able to infer from this that the user is driving a car. In summary, the *Context Interpreter*'s capabilities include:

- Context gathering: the *Context Interpreter* is able to collect context from *Context Providers*;
- Context aggregation: the *Context Interpreter* is able to aggregate context when needed. For instance, several *Context Providers* may be able to offer context of a certain user, one providing the user's location and the other providing the current time;
- Context inference: the *Context Interpreter* is able to infer context from other context(s). The inference rules are logic relationships between users' context and other information and they must be explicitly defined. We elaborate on this issue in Section 5.3.2.

Figure 16 depicts a possible configuration for the *Context Interpreter*. The *Context gatherer* is responsible for gathering context from the *Context Providers*. The *Aggregator* and *Inference Machine* are responsible for context aggregation and context inference, respectively. Both modules make use of the repositories to access information that supports aggregation and inferring activities. This example configuration shows three contexts being gathered by the *Context gatherer* (c1, c2 and c3). Contexts c1 and c2 are aggregated because they are associated with the same entity (e.g., time and location of a user). These aggregated contexts and the context c3 are passed to the *Inference Machine* in order to evaluate a certain inference rule. The result of this inference rule is the context c4, which can be provided to the *Repositories* or to the *Monitor* component.

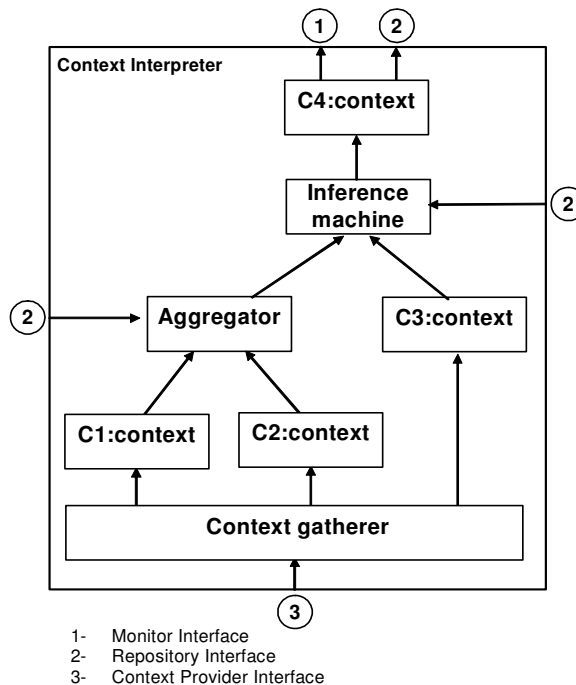


Figure 16 – Example configuration of the Context Interpreter

5.3.1 Information Provisioning Models of the Context Interpreter

The *Context Interpreter* uses three different models of information provisioning: *request-response*, *time-driven* and also the *event-driven model*. In the *request-response* model, the *Context Interpreter* provides context only on explicit request. In the *time-driven* model, the *Context Interpreter* is programmed to provide the context after specific time intervals. Finally, in the *event-driven* model, the *Context Interpreter* is programmed to provide the context only when the context value has been updated. In our design, the *access model type* is defined in the association of an *entity type* and a *context type*, as presented in Section 5.2.

Similarly, *Context Providers* can provide information following the three mentioned modes (*request-respond*, *time-driven* and *event-driven*). Therefore, the *Context Interpreter* decouples the platform from *Context Providers*, allowing the rest of the platform to use context information without worrying about different communication protocols that are necessary to communicate with *Context Providers*. Figure 17 depicts the different levels of information provisioning and the possible models of interaction.

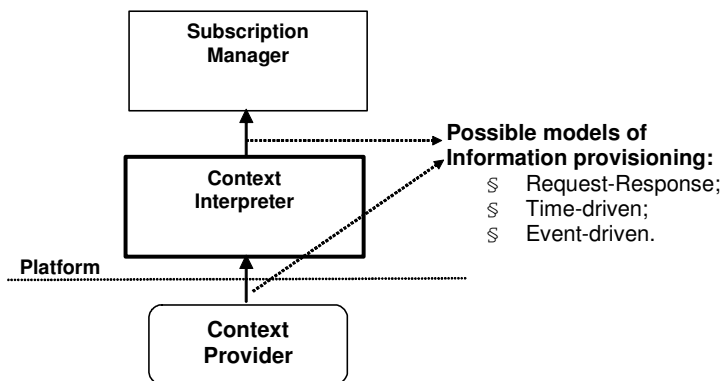


Figure 17 - Models of information provisioning

Figure 18 depicts the *Context Interpreter* providing information to the *Subscription Manager*, which is managing two subscriptions. Details about the *Subscription Manager* are given in Section 5.5.5.

The first subscription involves the context location applied for users *user1* and *user2*. The second subscription involves the context location applied for users *user2* and *user3*. It is defined in the platform that the *Access Mode* of the association of an entity *user* and context *location* (represented by the notation **user:location**) is *time-driven* (in the entity type registry, Figure 14). Therefore, the *Context Interpreter* must provide the context **user:location** in a *time-driven* way, for instance, every 2 minutes (highlighted "pushes" in Figure 18). However, the provision model for the *Context Provider* is the *request-respond* model. Thus, in order to obtain context, the *Context Interpreter* needs to explicitly request for it (highlighted "req" and "resp" in Figure 18).

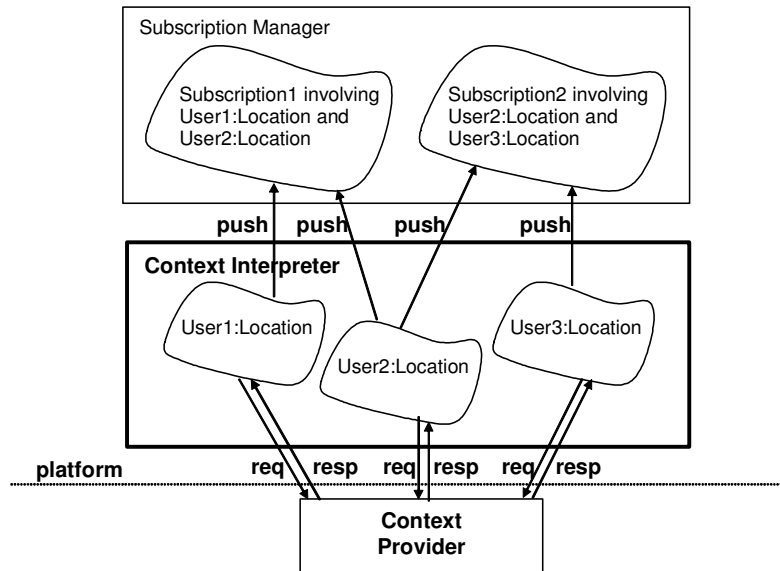


Figure 18- Example of context provisioning

5.3.2 Inferring Context

Context can be derived from other context(s). For example, the context **user.driving** is derived from consecutive verifications of context *location*. The context **user.working** can be derived from the user *location* and the *location* where he/she works. In order to be able to derive (infer) new contexts, it is necessary to define inferring rules in the *Context Interpreter*. Informally, we could say that an inferring rule defines logical relationships between information (context or not) in order to derive a new information that we are not able to directly sense from the environment. For example, an inferring rule looks like: if user's current location is inside his/her working place, we can infer that the user is working.

Inferring new contexts from existing ones, i.e., defining inferring rules can be very complex. We could greatly profit from Artificial Intelligence techniques to define and apply inference rules. The investigation and integration of such techniques are indicated for future work. Furthermore, we foresee that there will be third party context providers providing context at any desired level. This fact alleviates the platform from the burden of inferring complex contextual information.

Figure 20 depicts an example in which the context **user.working** is derived from the context **user.location** and information over the user's place of work and the location of the place of work (**building1.location**). The push model (time or event-driven model) is being used for both *Context Interpreter-Subscription Manager* and *Context Provider-Context Interpreter* interactions. Figure 19 shows the sequence diagram for this example of inferring context.

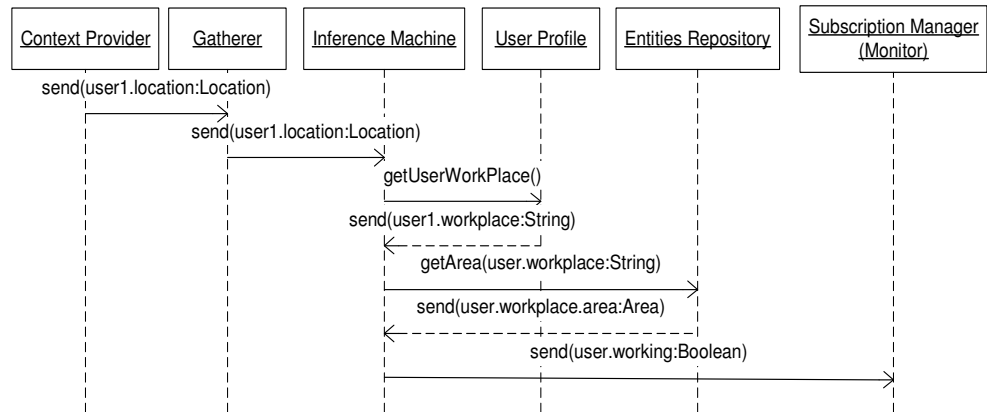


Figure 19 - Sequence of messages between modules for a context inference scenario

The *Context Gatherer* receives the context **user1.location** from the *Context Provider*. Following, the *Context Gatherer* forwards this context to the *Inference Machine*, which holds the inference rule that involves this context. This inference rule also involves the working place of the user and the working place area information, which are provided by the *User Profile* and the *Entities Repository* components, respectively. At this step, the *Inference Machine* possesses all the information necessary to perform the inference rule, which produces the context **user.working** (true or false).

The inferred context may be added by the subscription to redirect calls of *user1* to his secretary when he is working. We could even have a more specific redirection, for instance, to the room where *user1* is currently located.



Figure 20- Example of context inference

5.4 Repositories

The *Repositories* are the architectural components that contain and maintain the information represented in the *data entity model* (Section 5.2). As we will see in the next sections, *entity types*, *entity instances*, *action types* and *function types* are essential information to allow dynamic deployment of applications through the subscription language.

5.4.1 Entity Type Registry

The *Entity Type Registry* is the repository of entity types (and their attributes and context types) registered in the platform. Examples of entity types are *user*, *bakery*, *hospital*, *restaurant*, etc. Examples of *Attributes* are *MobileNature* (mobile or fixed), age, address, etc., and examples of *Context Types* are location, velocity, time, etc. It is possible to apply different kinds of context for different entity types. Velocity, for instance, is a context that may be applied to entity type *user* but may not be applied to *hospital* or *bakery*. The *Entity Type Registry* needs to keep track of all possible combinations of *context types* and *entity types*.

The platform offers a programmatic interface to allow *Entity types* and *Context Types* to be added (or modified) on demand, for example, by a platform administrator. Examples of operations of this interface are:

```
addEntityType(String name, AttributeType[] attr, ContextType[] cont)  
addContextType(String name, Value val)
```

The operation **addEntityType** adds an *Entity Type* with the specified name, list of attributes and list of context types. The types **AttributeType** and **ContextType** also have to be specified. The operation **addContextType** adds a *Context Type* with a name and value.

5.4.2 Function Type Registry

Functions are operations that perform a computation with no side-effects, i.e., it does not change the current status of the platform. Application subscriptions make use of *Functions* (in combination with *actions* and *data entities*) to express what they need from the platform. For instance, suppose that an application subscription needs to express (and check) whether two entities are close to each other. The platform should provide and/or support functions that perform this kind of operation.

Function Type Registry is the registry for information about the types of *Functions* supported by the platform. The characteristics of a *Function* are its goals (what it intends to do), the number and types of parameters it accepts and its return value. The platform retains as many *Functions* as necessary to perform its operations.

The primitive *function* **closeBy** for example, returns **true** if two or more entities are close to each other (for instance 20 meters) and **false** otherwise. This *function* only makes sense when two or more entities are involved (proximity of a single entity does not exist). Moreover, it only makes sense if the contextual information *location* is applicable for those entities.

Another example of *Function* is the **Match** function, which is able to say if the personal profile of a user matches with an other entity. **Match (user.user1, restaurant.LosPonchos)** returns **true** if user1 likes Mexican restaurants (LosPonchos is a Mexican restaurant) and **false** otherwise. Sophisticated approaches can be used in order to perform personalized services according to user preferences. Although dynamic adaptation of services according to user's preferences is an important issue in context-aware computing, we do not explore this issue in this work.

Other examples of functions are shown in Table 3:

Function	Explanation
Inside (entity1, entity2)	Verifies if entity1 is physically inside entity2. Entity2 must designate a geographical area.
List (entity, collection)	Makes a list of kind <head, collection> in which the first element is an entity and the second element is a collection.
Count(collection)	Counts the number of elements in a collection.
InRoad(entity1, entity.road1)	Verifies if entity1 (a user or a vehicle) is on entity road1.
IsTraffic(entity.road1)	Verifies if there is traffic jam on entity road1.
GetTrafficInformation(entity.road1)	Returns textual information about the traffic situation on entity road1.

Table 3 - Examples of functions

From the application point of view, the combination of functions **closeBy** and **Inside** can be used by application subscription to express that an *action* should be taken if **entity1** is **closeBy entity2** and **entity2** is **Inside entity3**.

The platform provides a set of embedded primitive *functions* (e.g., **closeBy**, **Inside**, etc). Moreover, it provides configurability of functions in the sense that new functions can be added on demand during platform run-time. Introducing this level of configurability greatly increases the usability of the platform.

Using *Web Services*, which is the technology chosen by the WASP project to implement the interactions of the platform with its environment, we were able to design a solution for dynamic deployment of *functions*. This solution involves the remote implementation of *function* as web service end-points. Furthermore, the platform provides an interface that allows the dynamic addition of the functions' signatures.

The following code represents the platform's interface to add functions:

```
interface FunctionRegistry
{
    addFunction(String name, int number_parameters, URL function_service_url);
    addFunction(String name, type[] parameters, URL function_service_url);
}
```

The method **addFunction** receives the name of the function, the number of parameters (or a list of parameters' types) and a URL where the Web Service endpoint is located.

In order to be executed by the platform, the implementation of the *Function* needs to provide the following interface:

```
interface Function
{
    ReturnValue execute(Value[] list);
}

class Value
{
    String name;
    String type;    // optional
    String value;
}

class ReturnValue
{
    String type;    // optional
    String value;
}
```

Therefore, a *Function* is executed by the platform by remotely calling the method **execute** of the function's web service with a list of parameters. The parameters are identified by their names, types and values. The **execute** method returns a value identified by its type and value.

The architectural element called *Parser* (Section 5.5.4) is responsible for performing syntactic and semantic check of application subscriptions. Since *functions* are basic operations used to write application subscriptions, the *Parser* component constantly checks the *Function Type Registry* in order to verify the correctness of *functions*.

5.4.3 Action Type Registry

Action Type Registry is the registry for keeping information about types of *Actions*. *Actions* are tasks performed in response to an enabling application subscription. Like *Functions*, characteristics of an *Action* are its goals (what it intends to do), the number and types of parameters it gets and its return value. However, *Actions* and *Functions* are semantically different. While a *Function* performs a computation with no side-effects, an *Action* typically has side-effects either on the application, users or platform. We will elaborate on this later in this chapter.

An example is the *action* **SendMessage** mentioned in the bakery example. This *action* should have the recipients of the message and the contents of the message: **SendMessage(entity.user.user1, "Bakery around. Don't forget to by bread.")**. The **SendMessage** *action*, in particular, affects directly the user via the underlying 3G Networks, i.e., the platform will use the message service of the 3G Networks to

send the message to the user. There are cases, however, in which the platform does not affect directly the user but instead, it affects WASP applications. Considering the bakery example but rather than receiving a text message, the user wants to see the closest bakeries on a map. A map service is not located in the platform but in a WASP Map Application (outside the platform). In order to show the bakeries on the map, the application gathers the location of the user and the locations of the bakeries surrounding the user from the platform.

Using the services of the platform, WASP Applications do not need to bother with how to gather this information (but it does need to know how to interact with the platform in order to expose its needs). Moreover, applications need to know the right moment to show the user the map since the user only wants to see it in a specific moment (only when there are bakeries close to him).

The sequence diagram in Figure 21 demonstrates the interaction between the WASP Applications and platform and Table 4 depicts the messages exchanged. Note that the *user* is not depicted in the Figure as well as the *Context Provider*.

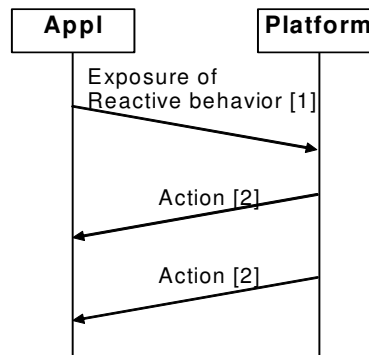


Figure 21 – Example of application-platform interaction

Message Number	Message Contents
[1]	Application subscription using the action NotifyApp() and the combination of functions CloseBy() and List() in order to gather the bakeries that are around John. The action NotifyApp() will be taken only when there exist bakeries around John.
[2]	List with the close bakeries: { John , { bakery1 , bakery10 }}

Table 4 - Bakery example

Examples of application subscriptions and more details of the WASP Subscription Language (WSL) are shown in Section 5.5.1.

Once the application has the locations of the user and the bakeries, it plots them in a map to the user. Open attributes are the attributes that the application is allowed to have access. Depending on the application, the access rights to attributes can change. It is possible to the application to be more specific choosing in advance the desired attribute, in this case location.

As already mentioned, actions can either affect applications, users or the platform itself. In the application map example, the action directly affected the application making the application to reconfigure the map according to the locations of the

user and bakeries. In the example in which the user gets a text message, the action directly affects the user. Actions can be used to change data in the platform. For instance, updating the *User Profile* and disabling another subscription could be both actions.

Other examples of actions are depicted in Table 5.

Action	Explanation
NotifyApp (expr)	It stands that the result of expr must be returned to the application that exposed this subscription.
UpdateUserProfile (entity.user.id, data)	Updates the User Profile of user entity.user.id with information data .
MessageON (entity.user.id)	Allows sending message to entity.user.id .
MessageOFF (entity.user.id)	Prohibits sending message to entity.user.id .
SendSMS (entity.user.id)	Sends a message to entity.user.id .

Table 5 - Examples of actions

There are situations, in which actions are mutually exclusive, i.e., they cannot be performed at the same time. For instance, the actions **SendMessage** and **MessageOFF** applied to the same user can generate a conflicting situation. For this reason, the platform needs to know which actions are conflicting and this information is made available in the action type registry. Conflicting situations are solved by the Coordinator component. We will elaborate on this in Section 5.5.6.

Similarly to *Functions*, *Actions* can also be added on demand during the platform run-time. The platform provides an interface that allows addition of actions' signatures. Moreover, those *actions* need to be remotely implemented as a Web Service.

The following code represents the platform's interface to add actions:

```
interface ActionRegistry
{
    addAction(String name, int number_parameters, URL action_service_url);
    addAction(String name, type[] parameters, URL action_service_url);
}
```

The method **addAction** receives the name of the action, the number of parameters (or a list of parameters' types) and a URL where the Web Service is located.

In order to be executed by the platform, the implementation of the *Action* needs to provide the following interface:

```
interface Action
{
    void execute(Value[] list);
}

class Value
{
    String name;
    String type;    // optional
    String value;
}
```

Therefore, an *Action* is executed by the platform by remotely calling the method **execute** of the action's Web Service end-point passing a list of parameters. The parameters are identified by their names, types and values. The **execute** method returns **void**.

5.4.4 Service Registry

The *Service Registry* component is responsible for storing, matching and retrieving of service profiles. Therefore, it deals with the *discovery and publishing of services* requirement, mentioned in Chapter 4, Section 4.2.4. The existing approach for service discovery in Web Services technologies is called UDDI (Universal Description, Discovery and Integration) [49]. UDDI specifications define a way to publish and discover information about web services offering [1].

Information provided in a UDDI registry consists of three components:

- *White pages* including address, contact, and known identifiers for web services providers;
- *Yellow pages* including industrial categorizations based on standard taxonomies; and
- *Green pages* including the technical information about services that are exposed by a web services provider. Green pages include references to specifications for web services.

There are, however, limitations in UDDI such as [38]:

- Lack of semantic interoperability, explicit semantic models to understand the queries and reason about the knowledge;
- Lack of ontology support. Service providers and service requestors might have different knowledge about a service. Service descriptions and service requests have to be understood and agreed upon the parties involved by means of an external party. A common ontology is necessary in order to facilitate an effective discovery process.

In order to overcome the lack of semantic interoperability and the lack of ontology support, enhancements in UDDI are being proposed in the literature, such as the Semantic Web UDDI [34]. The efforts of the WASP project towards an enhanced UDDI (called UDDI+ [38]) mainly include service discovery based on semantic information associated with the services. More specifically, the UDDI+ approach proposes semantic enhancement for context-aware services. In order to achieve

semantic interoperability of context-aware services, a context *ontology* is used to facilitate the context matching.

The UDDI+ works as follows: The service requestor creates a description of a virtual, desired service. Furthermore, the service requestor provides contextual information (location, temperature, etc). The matching then involves comparison of the requested service description with the registered ones by using the knowledge in the common service and context *ontologies*. Details about the UDDI+ architecture can be found in [38].

There are two integration levels of the UDDI+ in the platform:

- Low integration level: applications directly interact with the UDDI+, requesting services and providing context, which was previously gathered from the platform and
- High integration level: applications do not directly interact with the UDDI+ but only with the platform. The application-UDDI interaction is shielded by making reference of the desired services in the subscription added to the platform.

For instance, an application is interested in the services geographically close to a certain user. Using the first mentioned approach, the application would have to request the user location from the platform and then, request the UDDI+ for the services around that location. Using the second approach, the application would directly make the request through subscriptions. The *Subscription Manager* is responsible for interpreting the subscription, gathering the user location and requesting the component UDDI+ for the services around that location.

The initial goal is the integration of the UDDI+ to the platform supporting the first approach. The next step, which is integrating the UDDI+ supporting the second approach, is currently being developed by parallel works inside the WASP project.

5.4.5 *Entity Registry*

The *Entity Registry* is the component responsible for storing instances of entities that are neither services providers nor users. Examples of such entities are vehicles, buildings, rooms, roads, etc. It is extensively used by the *Monitor* module in order to parse and to perform application subscriptions that involve these entities.

5.4.6 *User Profile Registry*

The *User Profile Registry* is the component responsible for storing data about the user (instance of entity user and its profile). Significant facts can be collected directly from the user profiles. Knowing these facts enable context-aware applications to adapt to their users in many different ways, and to set different system behaviors. As already mentioned in Section 2.3.2, the information contained in the profile can be considered as contextual information in the sense that it describes the environment in which the users desire to operate. As such, it represents just a small part of the information domain of context-aware systems.

The *User Profile Registry* includes the following conceptual building blocks:

- **Identities:** ways to identify the user. A person can have more than one identity linked to the same user profile, where each identity is used in a different context. For example, one may get access via a username and password at home, whereas the IP address of the user' computer is used at work;
- **Characteristics:** aspects of a person that are completely objective and independent of the context, such as date of birth, sex, first name, etc;
- **Preferences:** personal subjective settings of a user towards the system that uses the user profile. Examples are the background color, sorting order of search results, number of search results to return;
- **Interests:** describe how a user is interested in certain concepts. An example is a user who is very interested in opera music;
- **Ratings:** ratings are explicit indications of how interesting a specific object is for the user;
- **History:** the history is a log of all actions taken by the user when using a system. All actions that somehow influence the user profile should be stored in the history of the user profile.

The abovementioned characteristics of a user profile are particularly interesting to allow adaptation of context-aware systems according to users' needs and preferences, which is a desirable behavior for the WASP platform.

Personalization services according to user preferences (user profile) can be requested by applications through application subscription. Currently, the platform offers an abstract primitive *function* to allow personalization, called **Match (entity.user, service)**. Although this *function* represents the possibility of service personalization request (by applications), it is just a naïve representation of what should be a complex definition of *functions* to allow a complete specification of personalization services. Exploring this subject is indicated for future work.

The *User Profile* is constantly checked by the *Subscription Manager* in order to interpret the subscriptions, especially when adaptive services are requested.

5.4.7 *ContextDB Registry*

The *ContextDB Registry* is responsible for preserving entities' contextual information over time. Applications can greatly benefit from keeping history of contextual information. In particular, keeping history of context is interesting to allow context inference based on past occurrences.

The *ContextDB* receives contextual information from the *Context Interpreter* in order to feed its contents and it is checked by the *Subscription Manager* in order to resolve application subscriptions that request any historical contextual information.

The following example applications are viable from the fact that the platform is capable of maintaining the *users' location* (an example of contextual information) history.

- Applications interested in retrieving business information related to points of interest, for instance, the profiles of users that visit a certain place including constraints of time;
- Applications that help on crime investigations. For example, it is possible to retrieve the history of the suspects' locations.

An application that is interested in knowing the speed (context) of the user in a given period of time (in order to check, for example, if the user really deserved a speed fine), benefits from the history of *users' speed*.

Information provisioning must strictly respect security and privacy policies. Aspects such as consent, locality, anonymity, pseudonymity and security must be addressed.

5.5 *Monitor*

The core of the platform architecture is the *Monitor* module. It is responsible for interpreting and managing the application subscriptions. Therefore, it tackles the requirements *Reactive behavior* and *Coordination among different applications*, mentioned in Chapter 4, Sections 4.2.2 and 4.2.3, respectively. In order to perform its operations, the *Monitor* makes use of the data available in the *Repositories* and the contextual information provided by the *Context Interpreter*.

Application-platform interactions are dynamically configured through addition of application subscriptions. By means of subscriptions, applications are capable of dynamically exposing their requirements to the platform. Application subscriptions are written in a descriptive language that allows them to expose their needs to the platform, during run-time. This language, coined the WASP Subscription Language (WSL), is discussed in Section 5.5.1.

The WSL manipulates *entities* (their *context* and *attributes*) and the combination of *actions* and *functions* in order to express the desired service. The fact that those elements are used through subscriptions makes the manipulation of the platform very flexible, since *entities*, *context*, *attributes*, *actions* and *function* can be added to the platform on demand (during run-time). The model that represents these entities and their context and attributes is defined in Section 5.2.

Figure 22 depicts the Monitor component. The subcomponents *Parser*, *Subscription Manager* and *Coordinator* are explored in Sections 5.5.4, 5.5.5 and 5.5.6, respectively.

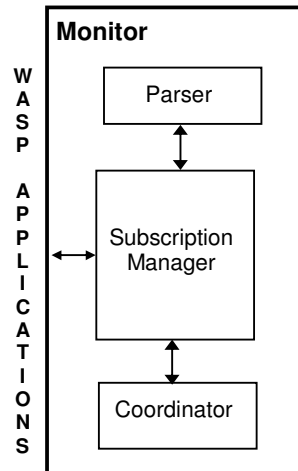


Figure 22 - Monitor Component

5.5.1 The WASP Subscription Language (WSL)

Application subscriptions provide the means to dynamically configure interactions between applications and platform. The platform model of operation (*request-response*, *time-driven* or *event-driven*) can be dynamically configured through subscriptions. In case of an *event-driven* platform behavior, the application subscription is the method used to “teach” the platform on how to react to a certain correlation of events. In case of a *request-response* platform behavior, it is the means by which applications make requests.

We have developed a descriptive language, not based on formal descriptions, to specify subscriptions. Initially, we identified two essential requirements with respect to elements in this language: (i) a way to specify the reaction of the platform and (ii) a way to correlate events, which will eventually trigger the specified reaction (Section 4.2.2 discusses these requirements). These two requirements resulted in the clauses **ACTION** and **GUARD** of the language. The moment the correlation of events specified in the **GUARD** clause turns **TRUE**, the action specified in the **ACTION** clause is triggered. The clause **GUARD** does not make sense when applied to the *request-response* model, which defines a simpler way of interaction: first request and immediately after the response.

Subscriptions can be either parameterized or not. Parameterization is necessary when the rule (subscription) is applied for a collection of entities. It would be cumbersome to write a subscription for each target entity. To allow parameterization, the clause **SCOPE** was introduced in the language.

Without this form of 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 newly introduced entities.

The non-parameterized subscriptions are composed by the basic structure **ACTION actions GUARD expr** for the event-driven model and **ACTION actions** for the request-response model.

From the necessity of filtering entities’ collections respecting a certain condition, 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.

Table 6 gives an overview of the clauses with their main objective. More details are given in the following paragraphs.

Clause	Function
ACTION-GUARD	Allows specification of an ACTION which is triggered when the GUARD turns true .
SCOPE	Allows parameterization of subscriptions.
SELECT	Allows the selection from a collection using filtering expressions (logical expressions).

Table 6 - Clauses of the WSL

The EBNF syntax of the language is depicted in Figure 24. While this concrete syntax is provided here, alternative representations could also be defined. For example, XML schemas (Appendix A) and UML Metamodels (Figure 23) can be considered as alternatives.

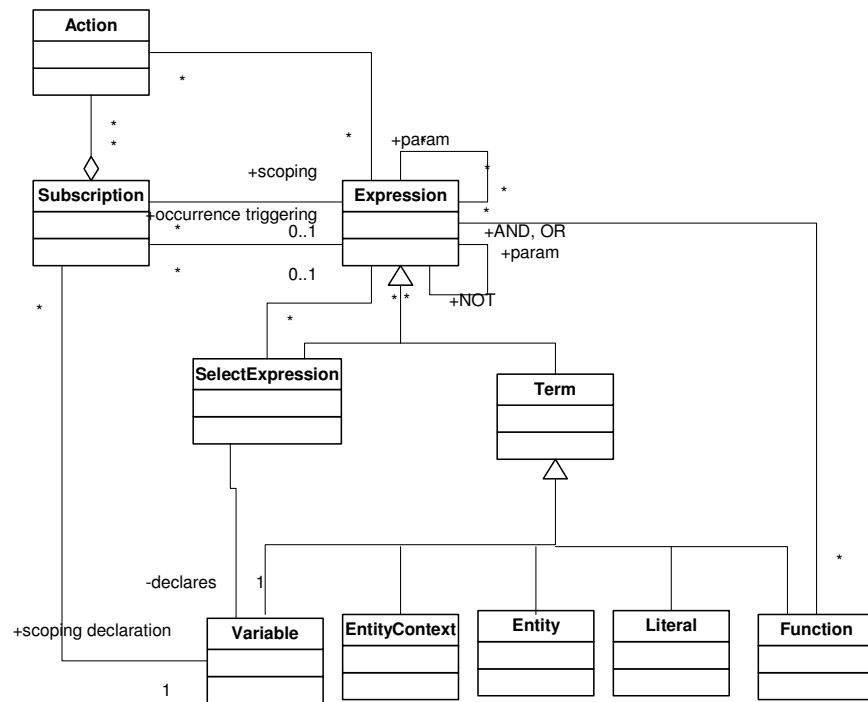


Figure 23- WSL syntax in UML representation

subscr	:	subscrparam subscraction
subscrparam	:	scope "{" subscraction "}"
subscraction	:	"ACTION" actions ["GUARD" expr]
scope	:	"SCOPE (" expr ";" " var ")"
actions	:	action ";" [actions]
action	:	ident "(" [expr ("," expr)*] ")"
expr	:	term unop expr expr binop expr select "(" expr ")"
select	:	"SELECT (" expr ";" " var ";" expr ")"
term	:	entitycontext entityattribute entities entity literal function var
entities	:	"entity." ident ".*"
unop	:	"NOT"
binop	:	"AND" "OR" ">" "<" "==" "<=" ">="
entitycontext	:	entity "." context entities "." context
entityattribute	:	entity "." attribute entities "." attribute
function	:	ident "(" [expr ("," expr)*] ")"
entity	:	"entity." ident "." ident var
attribute	:	ident
context	:	ident
var	:	ident
literal	:	stringliteral integerliteral
stringliteral	:	"" asciisequence ""
asciisequence	:	ascii [asciisequence]
ascii	:	A..Z 0..9
integerliteral	:	0..9 [integerliteral]
ident	:	identcharacter [ident]
identcharacter	:	ascii "_"

Figure 24- WSL syntax in EBNF

The following paragraphs detail the EBNF syntax depicted in Figure 24:

- An *action* has a name and possibly zero or more parameters, which are defined as *expressions*. The *actions* are all registered in the *Action Type Registry* (Section 5.4.3).
- An *expression* can be a *term*, a *unary operation* over an *expression*, a *binary operation* over two *expressions* and a **SELECT** expression.
- A **SELECT** expression is composed by the keyword "**SELECT**" applied to a collection restricting this collection some how. For example, **SELECT (entity.user.*; u; u.driving)** returns the collection of all users that are driving.

-
- A *term* can be the combination of an *entity* and *context*, the combination of an *entity* and *attribute*, a collection of *entities*, an *entity*, a *literal*, a *function* and a *variable*.
 - An *entity* is an instance of an *Entity Type*. For example, a *Restaurant* is an instance of *Entity Type* and *LosPonchos* is an instance of *Restaurant* (Figure 14). According to the defined syntax, the way to specify an *entity* is using the keyword "entity", the target *Entity Type* and the unique identification of the entity (**entity.entitytype.id**). The restaurant *Los Ponchos* is **entity.Restaurant.LosPonchos**. In order to give unique identification for the *entity*, it may be necessary to give more specific values since simple names are not good unique identifiers. However, we are using this approach to facilitate the demonstration of the language.
 - The word "entities" stands for a collection of all instances of a given *Entity Type*. **Entity.Restaurant.*** is the collection of all restaurants registered in the platform. If a more specific (filtered) collection is needed, it has to be explicitly done with the **SELECT** clause.
 - *Entitycontext* is the combination of an *entity* and one of its possible *contexts* or the combination of *entities* and one of their possible *contexts*. For example, it can be **entity.user.John.location**, which is referring to the location (context) of a specific user (John) or it can be **entity.user.*.location**, which is referring to the location of all users in the collection.
 - Similar to *entitycontext*, *entityattribute* is the combination of an *entity* and one of its possible *attributes* or the combination of *entities* and one of their possible *attributes*. For example, it can be **entity.user.John.address**, which is referring to the address (attribute) of an specific user (John) or it can be **entity.user.*.address**, which is referring to the address of all users in the collection.
 - Syntactically similar to an *action*, a *function* has a name and possibly zero or more parameters, which are defined as expressions. The *functions* are all registered in the *Function Type Registry* and the semantically difference between *functions* and *actions* is explained in Section 5.4.3.
 - A *literal* can be a *string literal* or an *integer literal*. A *string literal* is a sequence of ASCII characters and an *integer literal* is a sequence of integers.
 - An *identifier* is a sequence of ASCII characters and may be in combination with the symbol "_" (underscore).
 - The *unop* (unary) operator is the **NOT** logical operator.
 - The *binop* (binary) operators are the **AND** and **OR** logical operators and the comparison operators > (greater than), < (less than), == (equal to), >= (greater than or equal to), <= (less than or equal to).

The following paragraphs present the semantics of the WSL clauses.

The SELECT clause

The **SELECT** clause returns a collection of entities respecting a given filtering expression. Its abstract syntax is as follows:

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

A concrete example is **SELECT (entity.user.*; u2; u2.location.city == "Enschede")** which returns a collection of users located in Enschede at that given moment.

The ACTION-GUARD clause

The **ACTION-GUARD** clause defines an action (or actions) that should be triggered either immediately (request-response model) or in consequence of a correlation of events (clause **GUARD** is present). 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; (Inside(entity.user.John, c) AND
              (c.location.city == "Enschede")))
              // list of cinemas, where John is located, inv: 0 or 1
        )>0
      )
    )
```

This subscription sends a message to user John if and only if, John is inside a cinema and the cinema is located in Enschede. The **SELECT** clause is used to select a collection of cinemas in Enschede where user John currently is. This resulted collection has 0 or 1 element (either the user is in one or in zero cinemas). If the user is in one, an advertisement will be 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. The **ACTION-GUARD** is 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>
}
```

A concrete example is the scenario "Send an advertisement to every user in Enschede when he/she is inside the movies (also 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; (Inside(u,c) AND
      (c.location.city == "Enschede")))
      // list of cinemas, where u is located, inv: 0 or 1
    )>0
  )
}

```

As already mentioned, the **SELECT** clause returns a collection of users located in Enschede in that given moment. For each of these users (named "u" by the scope clause), the **ACTION** clause will be checked.

5.5.2 The Subscription State Machine

In order to govern the life cycle of a subscription, we represent it in a state machine, as depicted in Figure 25. The state *Initial* represents the moment when the subscription is added to the platform; the state *False* represents the state in which the subscription's **GUARD** clause is **false** and state *True* represents the state in which the subscription's **GUARD** clause is **true**.

A subscription takes a transition (changes state) when its **GUARD** clause changes value (from **false** to **true** and vice-versa). We have defined that the *actions* (in the **ACTION** clause) are only performed when subscriptions take the transitions from state *Initial* to *True* and from *False* to *True*. This way, when the correlation of events turns **true** (transition to state *True*), an *action* is triggered. If the correlation of events stay **true** (stay in state *True*) or turn **false** (transition to state *False*), no *actions* are performed.

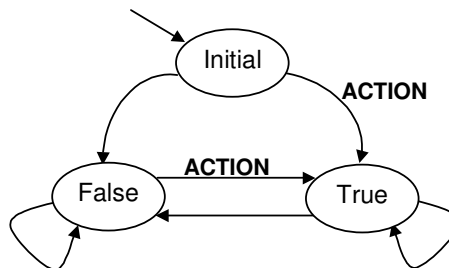


Figure 25- Subscription state machine

In the aforementioned cinema example, John will receive the advertisement once the platform detects he is inside the movies (transition from state *False* to *True*). While John stays in the movies (state *True*), no actions are triggered (for this subscription). John, eventually, will leave the movies and the state machine goes to the *False* state again (transition from state *True* to *False*). The state *Initial* was introduced because the subscription can be included in the platform during the platform run-time. John could be already in the cinema at that moment in time. In this situation, the state machine goes from state *Initial* to state *True* and the action is triggered. If the subscription is included (added to the platform) when John is

not in the cinema, the state machine goes from state *Initial* to *False* and no actions are triggered.

5.5.3 Scenarios

The applicability of the WSL has been demonstrated by a number of application scenarios. The flexibility of the language can be seen from the variety of applications that it is capable of covering.

We have selected three scenarios in which we present sequence diagrams to show the messages exchanged between users, applications, platform and context provider. These scenarios involve a *Taxicab company* application, a *Follow Me* application and a *Policemen* application. Furthermore, Table 10 briefly presents some additional scenarios and the corresponding application subscriptions.

Taxicab company application

The user requests a taxicab using the taxi cab company's web site (Taxicab application which is considered a specific WASP application). The taxicab application, supported by the WASP platform, is able to find the location of the user and the taxicabs in the vicinity of the user. With this information, the application selects the closest taxi available to pick the user up. The application also asks the platform to inform the user when the taxicab has arrived to his/her current location.

Figure 26 shows the sequence diagram of this scenario and the Table 7 depicts the interaction messages between the parties involved.

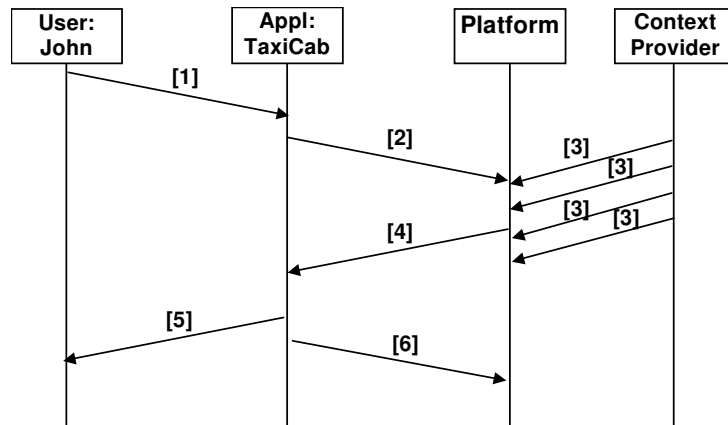


Figure 26- TaxiCab Application

Message Number	Message Contents
[1]	"I need a taxicab."
[2]	<pre> ACTION NotifyApp(List(entity.user.John.location, SELECT (entity.taxicab.*; tc; ((CloseBy (tc, entity.user.John, 3000)) AND (tc.company = "ABC"))))); </pre>
[3]	John's location and taxicabs' locations
[4]	Location of John and the taxicabs around him in format <head, list>: {john's location, {cab1234, cab456, cab1789}}
[5]	"It will be there in 5 minutes."
[6]	<pre> ACTION SendSMS(entity.user.John, "Your taxicab has arrived."); GUARD CloseBy(entity.user.John, entity.taxicab.cab1234, 50) </pre>

Table 7 - Taxicab scenario

The **SELECT** clause returns the collection of taxicabs that belong to company "ABC" and is less then 3000 meters far from John.

List is a function previously registered in the *Function Type Registry*. It takes two parameters, the first is the head and the second is the list to be appended to that head, and forms a list out of them with format <heard, {list}>. In this case, John's location is the first parameter and the list of taxis is the second.

NotifyApp is an action previously registered in the *Action Type Registry*. It returns the list with John's location and list of taxicabs to the application.

Tourist map application (Follow Me)

A user on vacations wants to have the map of the city he/she is currently in and wants to see the points of interest of the city placed on the map while he/she walks.

Figure 27 shows the sequence diagram of this scenario the Table 8 depicts the interaction messages between the parties involved.

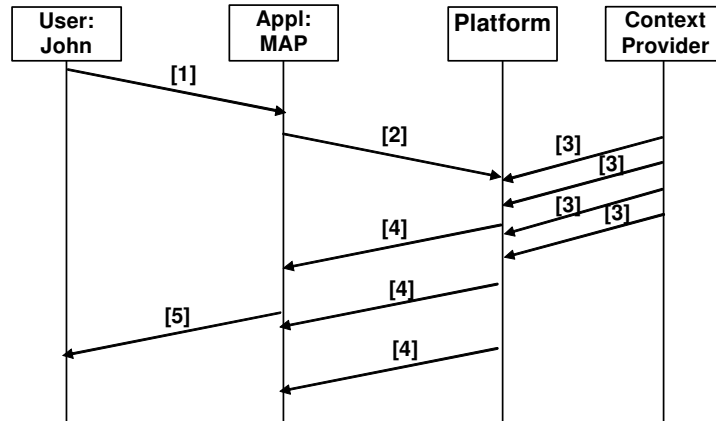


Figure 27- Follow Me application

Message Number	Message Contents
[1]	"I want to have the Follow me application."
[2]	<pre> ACTION NotifyApp(List(entity.user.John.location, SELECT (entity.POI.*; p; (CloseBy (p, entity.user.John, 200))))); GUARD OnEvery(180); </pre>
[3]	John's location and points of interest around him.
[4]	Location of John and the points of interest around him in format <head, list>: {john's location, {museumA, museumB, restaurantC, churchD}}
[5]	Refresh map with user's location and points of interest.

Table 8 - Follow Me scenario

The platform notifies the application every three minutes (**OnEvery(180)**) the new location of the user and the location of the points of interest around him.

Policemen application

Every policeman, for security reasons, should be aware of the colleagues close to him/her. This application shows on a map, in the policeman device, the colleagues around him/her.

Figure 28 depicts this scenario for three policemen that are close to each other the Table 9 depicts the interaction messages between the parties involved.

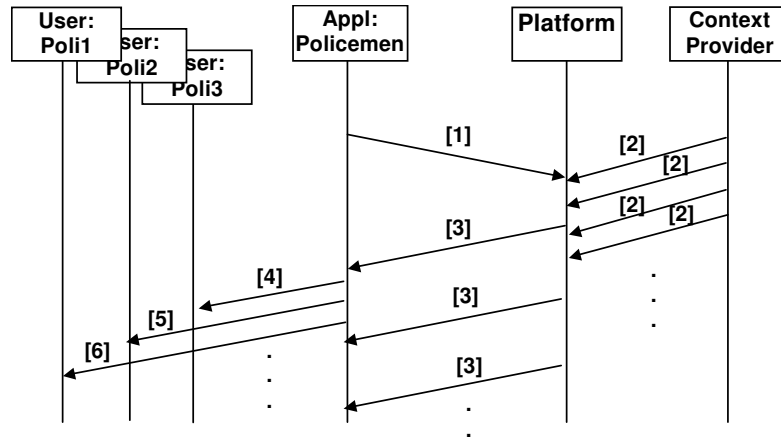


Figure 28- Policemen application

Message Number	Message Contents
[1]	<pre> SCOPE (SELECT (user.policeman.*; p2; p2.working); p) { ACTION NotifyApp(List(p, SELECT (entity.policeman.*; p3; (CloseBy (p, p3, 300) AND p3.working)))); GUARD OnEvery(900); } </pre>
[2]	Location of the policemen
[3]	Location of the policemen that are working and the colleagues working around each of them. {poli1 {poli2, poli3}}, {poli2 {poli1, poli3}}, {poli3 {poli1, poli2}}.
[4]	Refresh map with the location of the colleagues poli1 and poli2.
[5]	Refresh map with the location of the colleagues poli1 and poli3
[6]	Refresh map with the location of the colleagues poli2 and poli3

Table 9 - Policeman application scenario

The **SCOPE** clause defines that the rule is applied to all policemen that are currently working. The rule defines the application must be notified every 15 minutes about the location of the policemen and the colleagues working around them.

Table 10 shows some other additional examples of common user's scenarios and the respective(s) application subscriptions written in WSL.

User's scenario	Application Subscription
Send me a reminder of doing the shopping when passing close by a supermarket.	ACTION SendSMS(entity.user.John, "Do not forget to do the shopping.") GUARD Count (SELECT (entity.supermarket.*; s; CloseBy(s, entity.user.John, 100))) > 0
Send me a message when Alice is close by me.	ACTION SendSMS(entity.user.John, "Alice is close by.") GUARD CloseBy(entity.user.Alice, entity.user.John, 50)
Set my mobile OFF when I am driving.	ACTION MobileOFF(entity.user.John) GUARD entity.user.John.driving
	ACTION MobileON(entity.user.John) GUARD NOT (entity.user.John.driving)
Set my mobile OFF when I am in the cinema.	ACTION MobileOFF(entity.user.John) GUARD Count (SELECT (entity.cinema.*; c; Inside(entity.user.John, c))) > 0
I want to be informed of traffic jams in the roads I am currently driving.	ACTION SendSMS(entity.user.John, "Traffic Jam!" + GetTrafficInformation (SELECT (entity.Road.*; r; (InRoad(entity.user.John,r) AND IsTraffic(r)))) GUARD Count (SELECT (entity.Road.*; r; (InRoad(entity.user.John,r) AND IsTraffic(r))) > 0

Table 10 - Different example scenarios

5.5.4 Parser

The parser component is responsible for verifying if the subscriptions are syntactically and semantically correct having as a reference the syntax of the WSL depicted in Figure 24. As soon as the application subscription is received by the Subscription Manager, it is forwarded to the parser component to be resolved. In order to check the semantics of the subscription, the parser makes extensively use of the repositories *Entity Type*, *Function Type*, *Action Type* and the instances repositories *User Profile Registry*, *Entity Registry* and *Service Registry*.

The result of the parsing is a tree composed by the primitive elements of the WSL. The following subscription has its parsed tree depicted in Figure 29.

```
SCOPE ( SELECT (user.policeman.*; p2; p2.working); p )
{
  ACTION
  NotifyApp(
  List(
  p,
  SELECT (entity.policeman.*; p3;
  (( CloseBy (p, p3, 300) AND
  p3.working )
  )
  )
  )
  );
  GUARD
  OnEvery(900); }

```

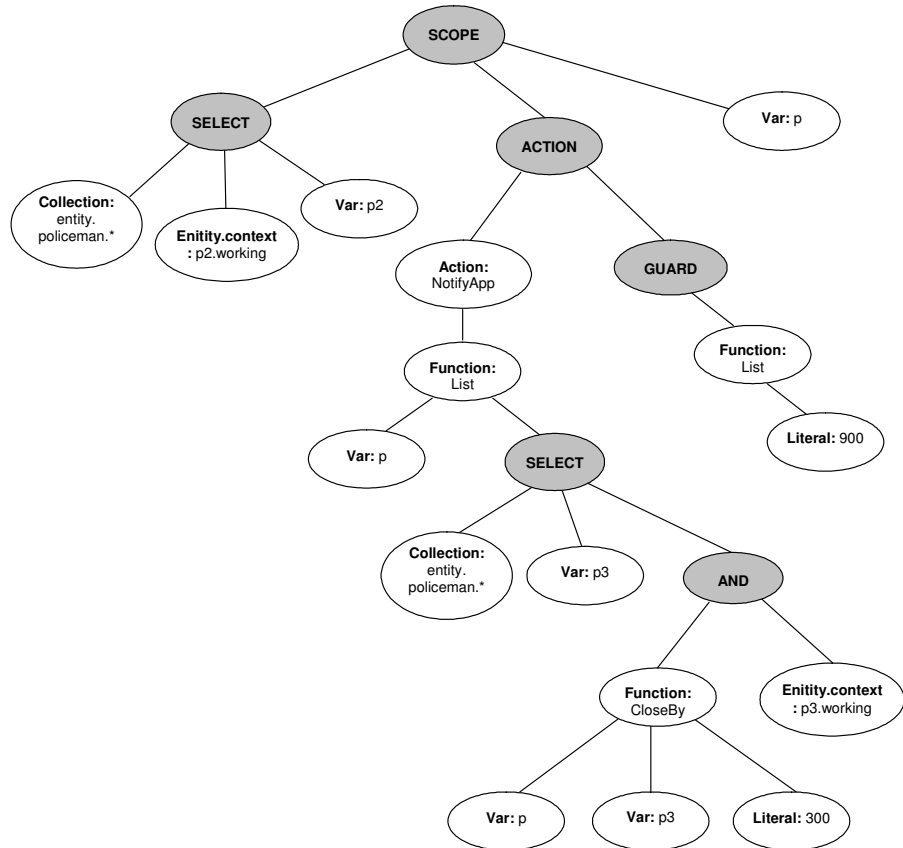


Figure 29 - Example of a parsed subscription

There are two levels of semantic checking:

- A model checking level using the platform entity metamodel as showed in Figure 14. At this level, the parser verifies the existence of the entity types and the combination of context type and entity type. For instance, the combination **entity.restaurant.driving** should give an error because the context **driving** is not applicable to **entity.restaurant**. The same holds for the context **velocity** when applied to any non movable entity. Moreover, the parser needs to verify the semantics of the *Functions* and the *Actions*. Specific functions and actions have specifically defined number and types of parameters. For instance the functions **IsTraffic()**, **InRoad()** and **GetTrafficInformation()** are specific functions for dealing with entity **entity.road**. **IsTraffic(entity.user.John)** should not be accepted because John is not an **entity.road** and therefore, he does not have traffic jams. The same sort of verification must be done with actions. For instance, the action **SendSMS** should always have at least one entity user as a parameter (recipient addresses).
- An instance checking level using the instance repositories to check the existence of the entities (final instances). If an application subscription uses entities **entity.user.John**, **entity.user.Alice**, **entity.restaurant.LosPonchos**, **entity.building.Informatica**, the parser needs to check the existence of such entities in the platform. In order to check the existence of users John and Alice, the parser checks the *User Profile* repository. In order to check the existence of the entity restaurant LosPonchos, the parser accesses the *UDDI+*, and in order to check the existence of the building Informatica, the parser examines the *Entity* repository.

5.5.5 Subscription Manager (SM)

The SM provides an API for manipulation of the application subscriptions. This API allows applications to add, remove or update subscriptions:

```
interface ApplicationPlatform
{
    SubsIdent addSubs (subs: String)
    void removeSubs (subs:SubsIdent)
    void updateSubs (id:SubsIdent, subs:String)
}
```

Moreover, applications should provide the **Notify** interface, so that, the platform is able to send data (results of actions) back to applications when this is requested. Results of actions, in this case, can be collections, an entity, an **entity:attribute**, an **entity:context**, a structure with header and a collection, a collection of collections, or just a string notification. A collection can be collection of entities, collection of **entity:contexts** and collection of **entity:attributes**:

```

interface Application
{
    void Notify (id: Subslid, data: <collection>)
    void Notify (id: Subslid, data: <collections>)
    void Notify (id: Subslid, data: <head, collection>)
    void Notify (id: Subslid, data: <head, collections>)
    void Notify (id: Subslid, data: entity)
    void Notify (id: Subslid, data: entity:context)
    void Notify (id: Subslid, data: entity:attribute)
    void Notify (id: Subslid, data: string)
}

```

Figure 30 depicts the internal view of the Subscription Manager.

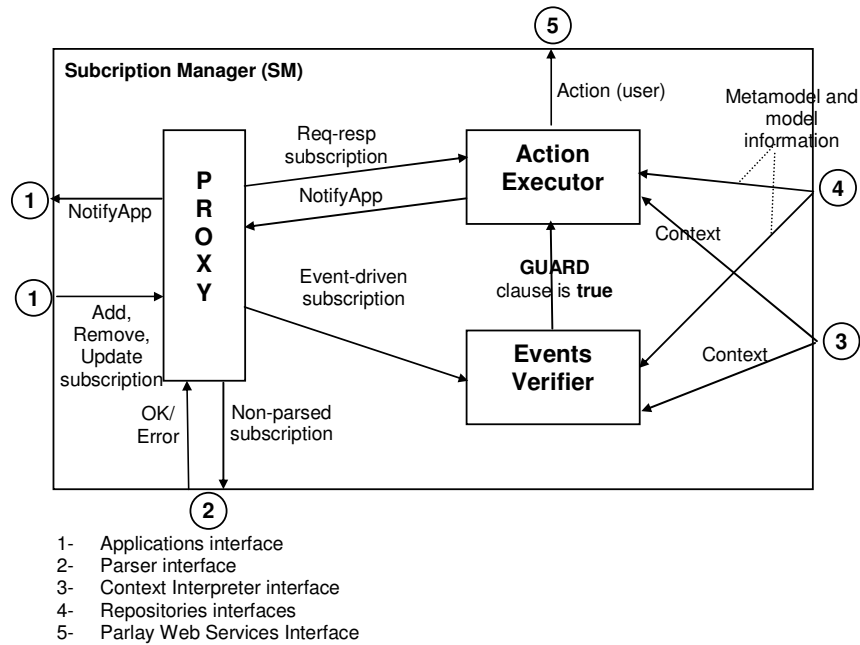


Figure 30- Subscription Manager

In order to perform the actions that directly affect the user, the platform needs to use the services provided by the underlying network. The WASP platform accesses the services of the underlying 3G Networks via the Parlay Web Services API [35]. This API allows access to the actual user’s context, playing the role of a *Context Provider*, and allows the user to be directly affected by some action (**SendSMS** or **MobileOFF**), playing the role of the enabling underlying network. Therefore, the SM must have knowledge of part of the Parlay Web Services API. This part refers to the services that allow the user to be directly affected by some action (**SendSMS**, for example). It is responsibility of the *Context Interpreter* component to handle the Parlay Web Services API regarding context provisioning. The definition and presentation of the API goes beyond the scope of this work and, therefore, will not be explored in this report.

An application subscription is parsed (sent to the *Parser* component) when added to the platform (**AddSub** operation) or when it is modified (**UpdateSubs** operation). In case of mistakes (syntactic or semantic), exceptions must be raised. The application should be able to check the exception.

Once the subscription is parsed and it turns to be syntactically and semantically compliant, the next step is to verify if the interaction is a request-response or an event-driven interaction. This is done by checking the existence of the **GUARD** clause. If the **GUARD** clause does not exist, it is a request-response interaction; otherwise, it is an event-driven interaction.

If it is a *request-response* interaction, the SM has to immediately resolve the subscription and trigger the specified *Action*. In order to resolve a subscription, the SM has to examine different elements in the platform, such as the *User Profile*, the *UDDI+*, *Entity Repository* and *Context Interpreter*. For instance, to resolve the piece of code **entity.user.***, the SM select all instance users in the *User Profile*. For the piece of code **SELECT (entity.user.*; u2; u2.location.city == "Enschede")**, the SM selects all the users in the *User Profile* whose current location is in the city called Enschede. To resolve the function **CloseBy (entity.user.John, entity.user.Alice, 100)**, the SM has to find out the current location of John and Alice from the *context interpreter* and then it has to check if they are less than (or equal to) a 100 meters far from each other. The sequence diagram depicted in Figure 31 shows the messages exchanged between the users, application and the SM elements in order to perform a *request-response* subscription.

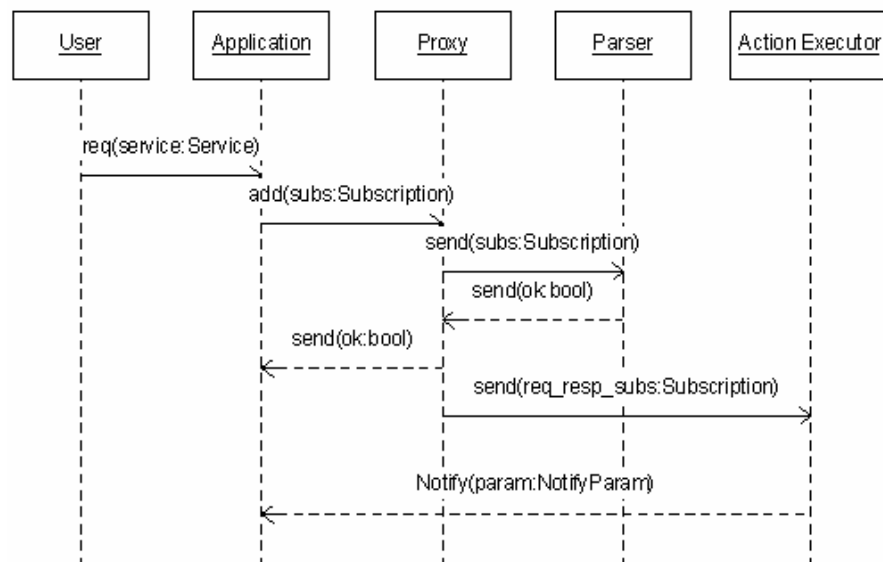


Figure 31 - Sequence of messages for request-response subscription

In case of an *event-driven* interaction, the SM needs to constantly check if the correlation of events defined in the **GUARD** clause is true. When the correlation of events turns true, the action is triggered just like if it was a *request-response* model of interaction. Figure 32 shows the messages exchanged between the users, application and the SM elements in order to perform an *event-driven* subscription. *Events Verifier* component will only send an action to be executed in the *Action Executor* component if the **GUARD** turns **true**.

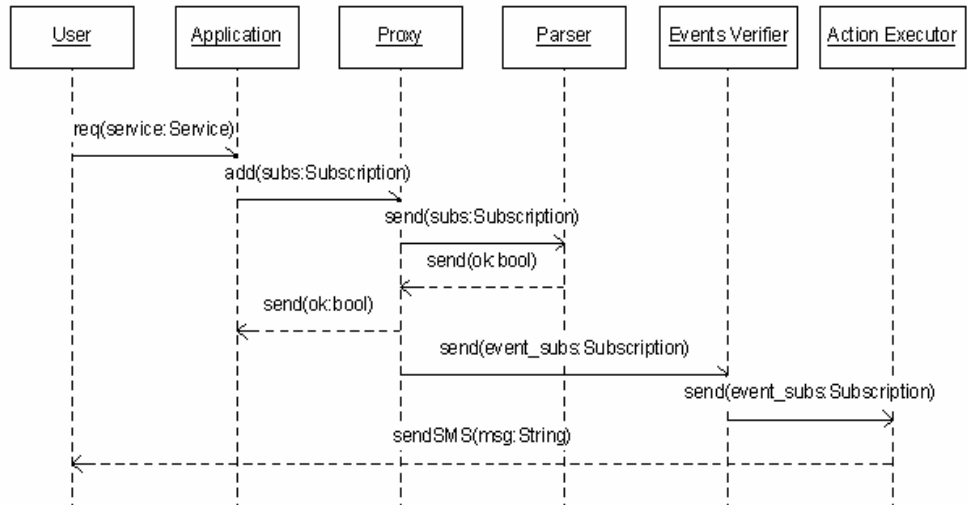


Figure 32- Sequence of messages for an event-driven subscription

In case of an *event-driven* model of interaction with **SCOPE** clause and with **GUARD** clause involving the scoping variable, the SM needs to constantly check the collection retrieved within the scope clause (it may change over time) and then verify the **GUARD** clause for each of the elements defined in this collection.

In the following example, there is a **SCOPE** clause that defines the collection of users (the ones that are in Enschede) with a **GUARD** clause involving the scoping variable (u). Therefore, the SM constantly checks the collection retrieved by the **SCOPE** clause because this collection may change since users may leave and arrive in Enschede. For each user in this collection (declared as variable “u”), the SM constantly checks whether the correlation of events defined in the **GUARD** clause holds in order to trigger the *action* similarly to the request-response interaction model.

```

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; (Inside(u,c) AND
      (c.location.city == "Enschede")))
      // list of cinemas, where u is located, inv: 0 or 1
    )>0
  )
}
  
```

5.5.6 Coordinator

This component is responsible for coordinating the conflicting application subscriptions. A subscription is conflicting with another subscription when the actions involved on it are mutually exclusive and they affect the same entity *user* and **GUARD** is the same for both subscriptions. For example, suppose the following two subscriptions are added to the platform:

Subs 1:
ACTION
 MobileOFF (entity.user.John)
GUARD
 entity.user.John.driving

Subs 2:
ACTION
 MobileON (entity.user.John)
GUARD
 entity.user.John.driving

Both define the same **GUARD (entity.user.John.driving)** and the actions **MobileOFF** and **MobileON** (conflicting actions) are applied to the same user (**John**). In the *Action Type Registry* should be specified which actions are conflicting and, consequently, the *Coordinator* is able to verify which subscriptions are potentially conflicting. A possible solution to solve the conflicting problems, is to have a priority queue among the conflicting actions, in the *Action Type Registry*. For instance, the action **MobileOFF** could have greater priority than **MobileON**. Therefore, using the given example, when John is driving, only the action **MobileOFF** is triggered.

6 Implementation

The goal of this chapter is to report the development of the prototype, which was implemented for demonstrating and validating some of the concepts we have proposed during the design of the platform architecture.

This chapter is structured as follows: Section 6.1 briefly states the approach we have chosen to implement the prototype. Section 6.2 reports how the *platform-application* interaction is implemented and gives special attention to the WSL parser. Section 6.3 discusses the implementation of the *platform-context provider* interaction. Section 6.4 presents three demonstrative scenarios and, finally, Section 6.5 presents some concluding remarks.

6.1 Approach

The main objective of the prototype is to demonstrate the concepts with respect to the *application-platform* interactions, as reported in Section 6.2. For that, we have implemented the following parts of the architecture:

- The platform interfaces to allow manipulation of subscriptions (addition, deletion);
- The platform *Subscription Manager* component which is responsible for managing application subscriptions added to the platform;
- The *Parser* to interpret subscriptions. The WASP Subscription Language (WSL) Parser is capable of reading application subscriptions in XML format and mapping them into Java classes, which are automatically compiled and executed during platform run-time;
- We have simulated some of the registries of the *Repositories* module, such as the *EntityType*, *ActionType* and *FunctionType* registries in order to support the *subscription manager* with knowledge of the entities involved in the application subscriptions; and

- A simple *Context Interpreter*, which gathers contextual information from a context provider (represented by the Location Simulator, Section 6.3).

Figure 33 depicts the abovementioned architectural elements in the WASP Platform architecture.

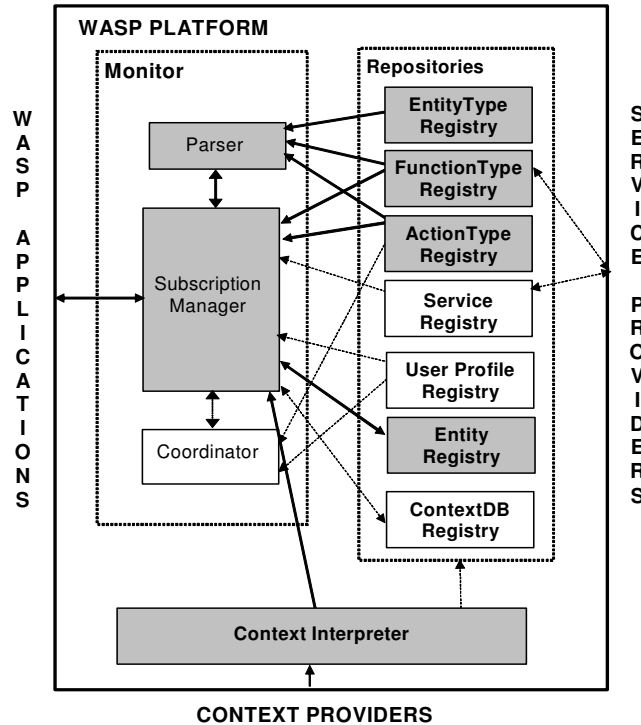


Figure 33 - Architectural elements implemented by the prototype

We have used Web Services technologies and Java language for implementing the prototype.

A web service provider is a software entity that offers web services [1]. A web service is a set of endpoints that operate on SOAP messages conveyed by internet protocols, such as HTTP, FTP and SMTP. Each endpoint is identified by a Uniform Resource Identifier (URI). A web service and its endpoints may be described in Web Services Description Language (WSDL). WSDL allows one to define the message types and message exchange patterns manipulated by web service endpoints, as well as the concrete means to interact with the web service endpoints, entailing concrete protocols for message exchange and the URIs that identify the web service endpoints.

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 [42] to automatically generate the WSDL file from Java interfaces. JAX-RPC is an API that provides support for mapping from WSDL to Java and vice-versa as part of the development of web service clients and endpoints.

Moreover, we have used the W3C's Document Object Model (DOM) [48] to parser application subscriptions written in XML format.

For purposes of demonstrating the usability of the prototype, we present three applications' scenarios: a proximity application scenario, an advertisement application scenario and a policemen application scenario.

6.2 Application-Platform Interaction

We have implemented the platform interface that allows application-platform interaction (insertion, deletion of subscriptions and application notification) in Java. The interface is available as a web service, such that, applications are able to remotely call the platform operations. Figure 34 depicts the interaction between WASP Application (service requester) and the WASP Platform (service provider).

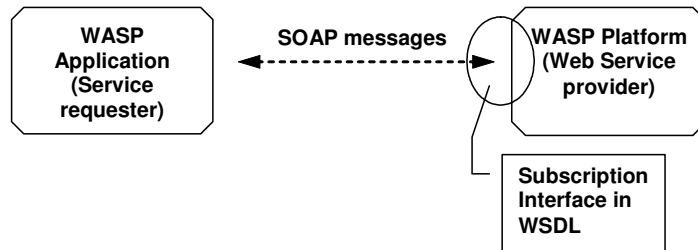


Figure 34 - Interaction between WASP Platform and WASP Application

Since Java classes are more readable than the corresponding WSDL description, we present the platform-application interface written in Java below and the corresponding WSDL description in Appendix B.

```
public interface SubscriptionInterface  
{
```

```
/**  
 * Adds a subscription.  
 */  
public String addSubscription(SubscriptionDescription sub_descr);
```

Add a subscription with
poling notification

```
/**  
 * Adds a subscription, specifying callback interface for notifications.  
 * @param sub_descr  
 * @param callback_uri  
 * @return  
 */  
public String addSubscriptionCallback(SubscriptionDescription sub_descr, String  
callback_uri);
```

Add a subscription with
callback notification

```

/**
 * Removes a subscription.
 * @param sub_id identifier of the subscription
 */
public void removeSubscription(String sub_id);

/**
 * Return a notification for a given subscription. Blocks until notifications are
 * available.
 * @param sub_id identifier of the subscription
 * @return the notification for subscription with identifier sub_id
 */
public SubscriptionNotification getNotification(String sub_id);
}

```

Removes a subscription

Gets a notification for a given subscription

The operation **addSubscription** returns a **String** which represents the identification of the subscription. For the operations **removeSubscription** and **getNotification**, the identification of the subscription needs to be provided.

The type **SubscriptionDescription** represents the **String** serialization of the XML subscription representation.

The following code defines the types used by the **SubscriptionInterface**:

```

public class SubscriptionNotification
{
    public SubscriptionNotificationElement[] notification_elements;
    public String subs_id;
}

public class SubscriptionNotificationElement
{
    // either text_element or entity_element will be nil
    public String text_element;
    public EntityElement entity_element;
}

public class EntityElement
{
    private String entityId;
    private String entityType;
    private EntityAttribute[] attributes; // attribute
    private EntityContext[] contexts;
}

public class EntityAttribute
{
    private String name;
    private String value;
}

public class EntityContext
{
    private String name;
    private String value;
}

```

Subscription Notification Type

Subscription Notification Element

Entity Element (with attributes and context)

The type **SubscriptionNotification** type defines an array of notification elements, identifying the subscription (**subs_id**) which they belong to.

The **SubscriptionNotificationElement** type defines that a notification element can be a **String** element (e.g., just an entity name) or an **EntityElement**, which is the serialization of an **EntityType** (Section 5.4.1).

6.2.1 WASP Application (client side)

Applications need to find the Subscription Interface Service (platform side) in order to add and remove subscriptions. The following code depicts how it would look in Java using JAX-RPC:

```
SubscriptionInterfaceServicePortType subs_service;  
subs_service = (new  
SubscriptionInterfaceServiceLocator()).getSubscriptionInterfaceServicePort();
```

Figure 35 depicts the basics activities of the WASP application in order to prepare and add an application subscription to the platform.

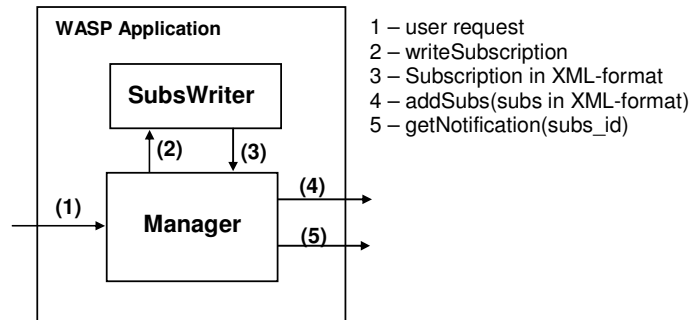


Figure 35 - Preparing a subscription to be added to the platform

Message (1) indicates a user request to the application. Message (2) depicts the application manager request to the SubsWriter to write a subscription in XML-format corresponding to the user's request.

In our approach, subscriptions are written by applications in XML format. The following code shows an example of WSL subscription and the corresponding XML representation.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Patricia Dockhorn
Costa (University of Twente) -->
<!--
Author: Patricia Dockhorn Costa (dockhorn@cs.utwente.nl)
Date: 15-06-2003
```

Corresponding WSL subscription description:

```
ACTION
  MobileOFF(entity.user.John)
GUARD
  entity.user.John.driving
```

Corresponding WSL
subscription

```
-->
```

```
<subscription xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="wsl.xsd">
```

```
<actions>
```

```
<action name="mobileoff">
```

```
<param>
```

```
<entity name="entity.user.id2"/>
```

```
</param>
```

```
</action>
```

```
</actions>
```

"ACTION" element of
a XML representation
of a subscription

```
<guard>
```

```
<entity_context entity_name="entity.user.id2" context_name="driving"/>
```

```
</guard>
```

"GUARD" element of
a XML representation
of a subscription

```
</subscription>
```

Once the subscriptions are written in XML format by the application (SubsWriter responsibility), they can already be syntactically validated in the application (client side) with the XML Schema presented in Appendix A (with help of a validation tool).

Having a syntactically valid XML representation of a subscription, applications are able to add it to the platform (remotely calling the operation **addSubscription**), represented by message (4) in Figure 35.

We have implemented a *polling* solution to look for new notifications, i.e., applications need to call the operation **getNotification** (message (5)) in order to receive subscriptions' notifications. Therefore, for each added subscription, a parallel process (a thread in Java) that looks for notifications is started. Once a **SubscriptionNotification** is arrived, the application freely manipulates the received information (e.g., plotting locations in a graph on the user's terminal).

An alternative solution to *polling* notifications from the platform is to implement the interface *Notify* on the application (to be available as a web service). This way, subscriptions must be added to the platform with a callback address (the URI that identifies the web service endpoint). Therefore, the platform is able to callback the application, informing it of possible notifications.

6.2.2 WASP Platform (server side)

When subscriptions are added to the platform they need to be parsed and executed. We have chosen a solution that parses a subscription written in XML format and map it into a Java class, which is automatically compiled and executed during the platform run-time. Figure 36 depicts the sequence of actions that are

taken when a subscription is added to the platform. The numbers define the sequence.

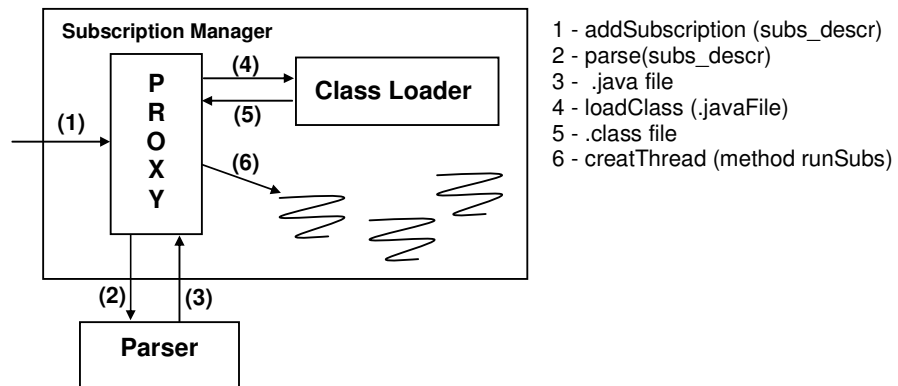


Figure 36 - Adding a subscription to the platform

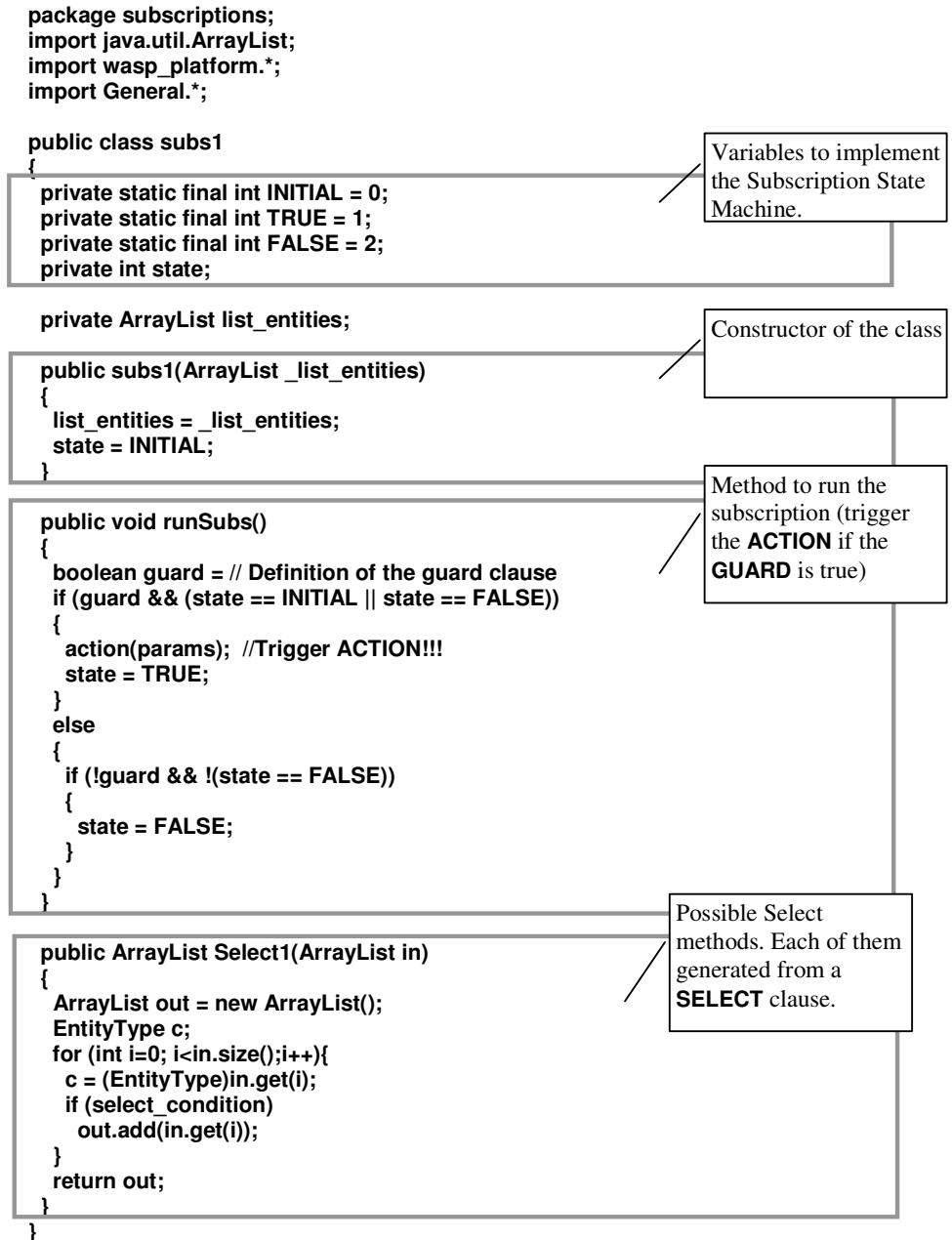
Immediately after receiving a new subscription, the XML document is forwarded to the parser (message 2), which creates a .java representation of that subscription (message 3).

The .java files are sent to the Class Loader, which is responsible for compiling the file (creating a .class) and loading the .class file into the Java Virtual Machine. Once the class is loaded, it is possible to manipulate the operations and attributes of this class by using Java reflection. Therefore, the method used to run the subscription (**runSubs**) is called in a parallel process (thread) created for each subscription.

6.2.3 The WSL Parser

In order to parse an XML document, we have used the W3C's Document Object Model (DOM) [48]. The DOM core defines a tree-like representation of the document, also referred as the DOM tree, enabling traversing the hierarchy of elements accordingly. Therefore, we are able to walk through the XML representation of a subscription reading the node elements and mapping them into correspondent commands in Java (using a recursive algorithm).

The structure of the generated .java file (class) is as follows (subscription without **SCOPE** clause):

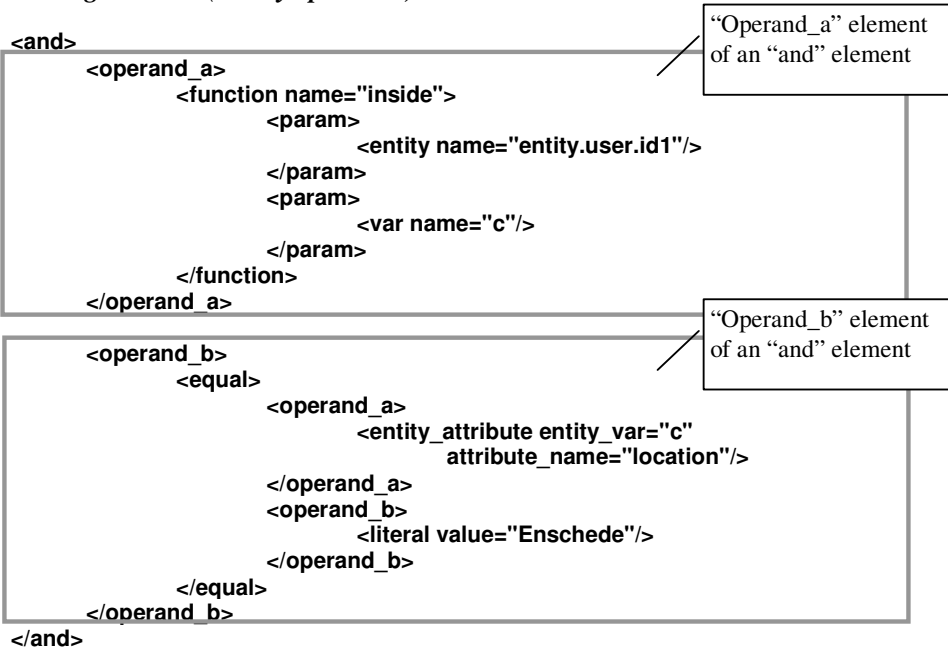


As mentioned in Section 6.2.2, each Java class representing a subscription has a **runSubs** method which is continually called (inside an infinite loop) from a thread. This method verifies the value of the **GUARD** clause (**true** or **false**) in order to decide whether to trigger the **ACTION**. This way, the **event-driven** model is implemented. In case of a request-response subscription, we define that the **GUARD** is true. In case of a **time-driven** subscription, an internal timer has to be implemented. This timer verifies the last time an **ACTION** was triggered and the current system time. If the time interval between the two is bigger than (or equal to) the interval defined by the **OnEvery** function, the **ACTION** should be triggered again.

The recursive algorithm to parse subscriptions verifies the node elements, checks the correspondence in the WSL Syntax, and returns a string representation of the

correspondent commands in Java, which are written in a .java file, using the mentioned structure.

Parsing an AND (binary operation) element



For example, consider the parsing of the node `<and>` (binary operation **AND**), described in the above written code. According to the WSL Schema (Appendix A), the node `<and>` has two children nodes, `<operand_a>` and `<operand_b>`. The recursive algorithm to parse the node (and translate it into java commands) looks like:

String parse (Node node)

```

{
  ...
  if (node = "AND")
    return "(" + parse(node.child(0)) + "&&" + parse(node.child(1)) + ")";
  if (node = "OR")
    return "(" + parse(node.child(0)) + "||" + parse(node.child(1)) + ")";
  if (node = "NOT")
    return "(" + parse(node.child(0)) + "!";
  ...
}

```

Annotations in the code:

- A box labeled `parse(operand_a)` points to the `parse(node.child(0))` call.
- A box labeled `parse(operand_a)` points to the `parse(node.child(1))` call.

Parsing the other existing nodes (functions, actions, entities, etc.) works similarly. The exit points of the recursion (leaves of the tree) are the nodes `<entity>`, `<entity_attribute>`, `<entity_context>`, `<entities>`, `<literal>` and `<var>`.

For example, consider the parsing of the node `<literal>`:

```

String parse (Node node)
{
    ...
    if (node = "LITERAL")
    {
        attribute_value = node.attribute(0);
        if (isNumber(attribute_value))
            return attribute_value;
        else
            return "\"" + attribute_value + "\"";
    }
    ...
}

```

Parsing the ACTION and FUNCTION elements

The `<action>` element (**ACTION** clause) declares a name and a list of `<param>` elements.

```

<action name="sendsms">
  <param>
    <entity name="entity.user.id1"/>
  </param>
  <param>
    <literal value="Coca-cola and film, a perfect combination!"/>
  </param>
</action>

```

Parsing an **action** is similar to parsing an **AND** node. The first recursion step parses the `<action>` element and the following steps parse the children nodes which are the parameters of the action (in this case, nodes `<entity>` and `<literal>`). The recursion stops in these nodes because they are leaves of the tree.

For this prototype, we have implemented internally in the platform some primitive actions such as **sendsms** and **notify**. In future versions, actions should be available as web services and the mapping needs to be slightly changed (we would have to invoke a service end-point). Using primitive actions, hard coded in the platform, the XML node `<action>` given as example, is represented in the Java file as follows:

```

General.action.sendsms (General.util.getEntityType ("id1"), "Coca-cola and film, a perfect combination")

```

This invocation (located in the **runSubs** method) triggers the action **sendsms** (from the static class **General.action**) passing an object (whose ID is "id1") and a string as parameters.

Parsing the element `<function>` is made similarly to parsing an `<action>`, except the functions are invoked from the class **General.function**.

The **General.function** and the **General.action** classes simulate the *Function Type Registry* and the *Action Type Registry* architectural elements, respectively.

Parsing the GUARD element

The `<guard>` element (**GUARD** clause) is just a logical combination (using logical operators) of `<function>`, `<entity>`, `<entity_attribute>`, `<entity_context>`, `<entities>`, `<literal>` and `<var>`. Therefore, this element is similarly parsed as depicted in the `<and>` node example.

Parsing the *SELECT* element

The `<select>` element (**SELECT** clause) contains a select variable, a collection and condition involving this variable. For example, the following piece of XML code represents a **SELECT** clause, which selects a collection of cinemas whose locations are in Enschede.

```

    Select variable
<select var="c">
  Select collection
  <collection>
    <entities>entity.cinema.*</entities>
  </collection>
  Select condition
  <condition>
    <equal>
      <operand_a>
        <entity_attribute entity_var="c" attribute_name="location"/>
      </operand_a>
      <operand_b>
        <literal value="Enschede"/>
      </operand_b>
    </equal>
  </condition>
</select>
```

Each **select** clause encountered in the XML document is mapped in a *Select* method in the java class. This method filters out the desired entities (defined in the child element `<collection>`) respecting the condition (defined in the child element `<condition>`). The following Java code depicts how a generated Select method looks like:

```

public ArrayList Select1(ArrayList in)
{
  ArrayList out = new ArrayList();
  EntityType c; //select variable
  for (int i=0; i<in.size();i++)
  {
    c = (EntityType)in.get(i);
    if (c.location.equals("Enschede"))
      out.add(in.get(i));
  }
  return out;
}
```

Parsing the *SCOPE* element

The `<scope>` element (**SCOPE** clause) declares a variable and a collection. For each element of this collection (represented by the scope variable), the clause **ACTION-GUARD** must be performed.

```

<scope var="u">
  <collection>
    <entities>entity.user.*</entities>
  </collection>
</scope>
```

The structure of the Java class subscription is different when a **SCOPE** clause is present since the `runSubs` method should consider the scope collection (**ACTION-GUARD** should be performed for each element of the scope collection). Moreover, the scope variable is used not only inside the `runSubs` methods, but also inside the

Select methods. Therefore, the scope variable needs to be declared as a class attribute.

The implementation of the Subscription State Machine is also different with a **SCOPE** clause because it is necessary to hold a state for each element of the **SCOPE** collection. Considering the example in which a scope subscription defines that all entity **users** should receive an advertisement when inside a cinema (located in Enschede). If John enters the cinema, the **ACTION sendsms** is triggered and his **state** turns **TRUE** (and the **ACTION** is not triggered until his **state** goes to **FALSE** and **TRUE** again). But Alice, another entity **user** part of the scope collection, has not entered any cinema yet. Therefore, her **state** must remain **INITIAL**. To address this issue, the class implements an array of **states**, one for each element of the scope collection.

The structure of the Java class with **SCOPE** clause is as follows:

```

public class subs1
{
    private static final int INITIAL = 0;
    private static final int TRUE = 1;
    private static final int FALSE = 2;
    private int[] state;

    private ArrayList list_entities;
    private EntityType u;

    ...

    public void runSubs()
    {
        ArrayList scope_collection = //Definition of the scope collection

        ...

        for (int i=0; i< scope_collection.size(); i++)
        {
            u = (EntityType)scope_collection.get(i);
            boolean guard = // Definition of the guard clause (involving "u")
            if (guard && (state[i] == INITIAL || state[i] == FALSE))
            {
                action(params); //Trigger ACTION!!!
                state[i] = TRUE;
            }
        }
        ...
    }
    ...
}

```

Diagram annotations:

- Array of states (points to `private int[] state;`)
- Scope variable (points to `private ArrayList list_entities;`)
- Method to run the subscription with scope collection (points to `public void runSubs()`)

6.2.4 Web Services – Java implementation issues

We have used the JAX-RPC API to map our **SubscriptionInterface** written in Java to WSDL. This approach has facilitated exposing some interfaces of the platform as Web Services. However, there are shortcomings of this technology which are important to be mentioned:

JAX-RPC is not able to map a **<choice>** element of a XML schema to Java. This has limited us with respect to the serialization of the application subscriptions in XML-format. We were not able to define the WSL-XML-Schema (Appendix A) as a serializable type in the **SubscriptionInterface** WSDL, because the **<choice>**

element is used in the WSL-XML-Schema. This limitation has forced us to use an opaque type (String) to serialize subscriptions written in XML-format.

Furthermore, JAX-RPC is not able to map class hierarchies from Java to WSDL. Therefore, we had to create a serializable flat class (called EntityElement) which represents our entity class hierarchy. Figure 37 depicts the serializable Entity Element, whose attribute “Type” defines which class in the hierarchy this element represents.

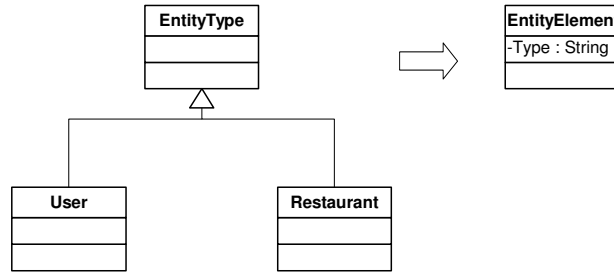


Figure 37 - Serializable EntityElement

6.3 Platform-Context Provider Interaction

In our prototype, the *Context Provider* is represented by the Location Simulator, implemented by a parallel work, inside the WASP project.

The Location Simulator interface is offered as a Web Service end-point, which allows the platform to remotely call the Location Simulator operations. Figure 38 depicts the interaction between the WASP Platform and the Location Simulator.

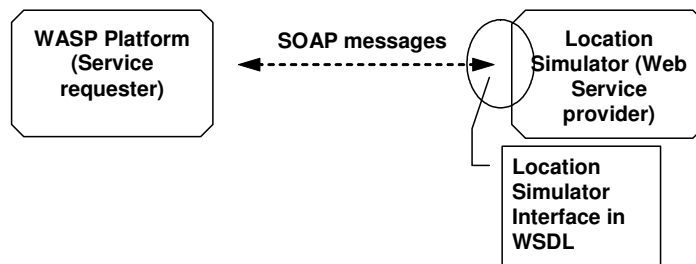


Figure 38 - Interaction between WASP Platform and Context Provider (Location Simulator)

6.3.1 Context Provider (server side)

The Location Simulator provides a graphical interface to monitor the current users in the system. Figure 39 depicts the monitor interface of the Location Simulator. This monitor allows us to (i) add new users, (ii) set their locations, (iii) define a route and (iv) define the user’s speed.

Figure 39 shows the user “John” in the center of Enchede. John is walking in the defined route (blue line).

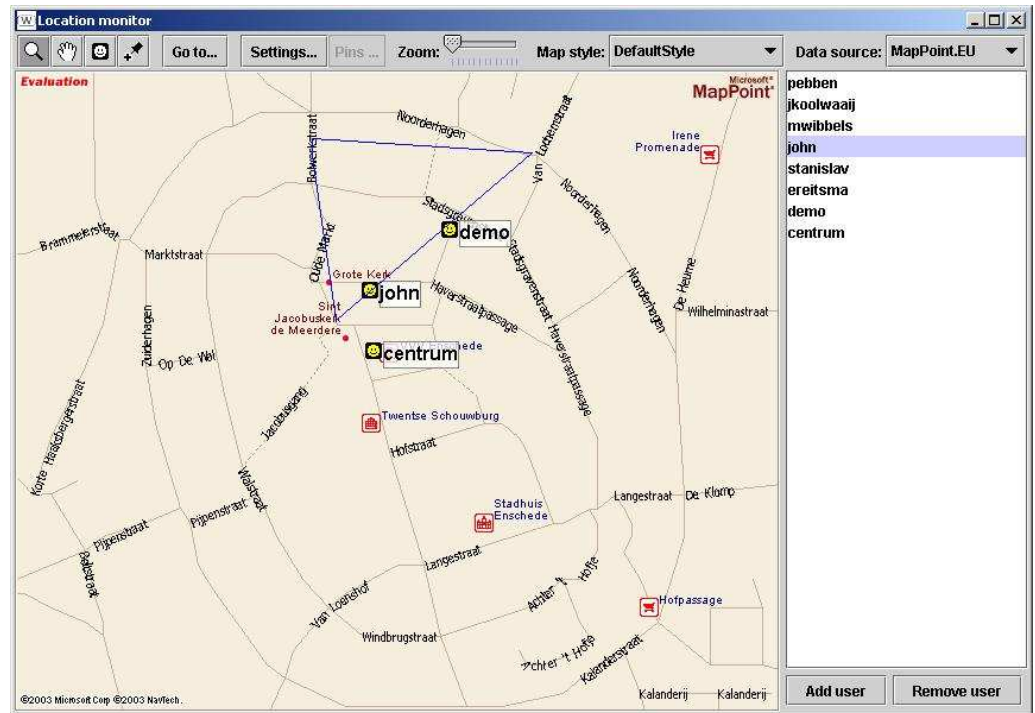


Figure 39- Location Simulator

The Location Simulator offers a Web Service end-point (WSDL file, Appendix C) which allows the access to the locations of the users that are plotted in the map. The following code represents the Location Simulator interface and the location type it manipulates.

```
public interface UserLocation
```

```
{
```

```
/**
```

```
 * Gets the user location in the location simulator.
```

```
 * @param the name of the user
```

```
 * @return LatLong
```

```
 */
```

```
 public LatLong getUserLocation(String name);
```

```
}
```

```
public class LatLong
```

```
{
```

```
 private double latitude;
```

```
 private double longitude;
```

```
}
```

Operation to get the user's location

Type that defines the user's latitude and longitude coordinates

The method `getUserLocation (String name)` offered by this end-point, is used to get the user location. It returns a `LatLong` type, which represents the latitude and longitude coordinates of the user.

6.3.2 WASP Platform (client side)

At the platform side, we have implemented a simple *Context Interpreter* that is able of gathering the users' locations from the Location Simulator.

The platform's *Context Interpreter* polls the Location Simulator (by invoking the **getUserLocation** operation) in order to gather information on the location of the platform's users. The *Context Interpreter* refreshes the list of entities, which is the simulation of the *Entity Registry* component, with the updated locations gathered from the Location Simulator.

This list of entities is also manipulated by the **runSubs** operation (in the java class that represents the subscription in the platform) in order to check the users' current context. In particular, the list of entities is used to check the value of the **guard** clause. Refreshing the list of entities with the updated contexts gathered from the Location Simulator guarantees that the **runSubs** operation always manipulates up-to-date locations.

6.4 Scenarios

We have demonstrated the prototype usage with several applications' scenarios. We discuss three of them in this Section: the policemen scenario discussed in Section 5.5.3; an advertisement scenario and a simple proximity scenario.

6.4.1 Policemen Scenario

The policemen scenario (Section 5.5.3) is a **time-driven** subscription with scope clause, which says that every working policeman, for security reasons, should see his/her colleagues that are 300 meters close by, in a map.

A WASP Application should be responsible for gathering the locations of the policemen from the platform and plotting them on the policemen's terminals. Moreover, the WASP Application needs to add the following subscription to the platform (XML representation of this subscription is given in Appendix D):

```
SCOPE (  
  SELECT (user.policeman.*; p2; p2.working);  
  p  
)  
{  
  ACTION  
  NotifyApp(  
    List(  
      p,  
      SELECT (entity.policeman.*; p3;  
        (  
          ( CloseBy (p, p3, 300) AND  
            p3.working  
          )  
        )  
      )  
    )  
  );  
  GUARD  
  OnEvery(10);  
}
```

The application invokes the **addSubscription** operation passing the XML representation of this subscription. A Java file is automatically generated, compiled and loaded into the Java Virtual Machine. Furthermore, the method **runSubs** of the generated file is constantly invoked. When the *Action notify* (NotifyApp) is called (every 10 seconds) by the platform, a list of notification

elements is created. As mentioned in Section 6.2.1, the application polls for notifications using the operation **getNotification**.

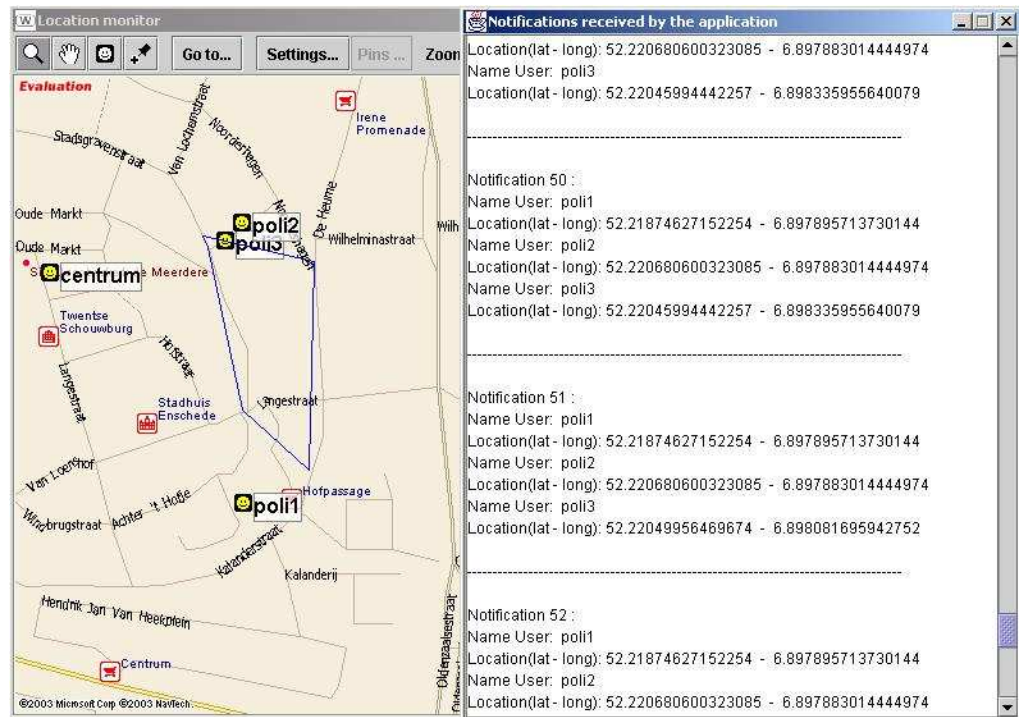


Figure 40 - Policemen scenario

Figure 40 depicts the policemen (poli1, poli2 and poli3) in the map and the Frame positioned by the side of the location simulator shows the notifications received by the application (sent by the platform) every 10 seconds. The notifications contain the locations of the three policemen. Only poli3's location is changing because poli3 is the only policeman in movement.

6.4.2 Advertisement Scenario

In this scenario, user John receives an advertisement message when he is inside a cinema and the cinema is located in Enschede. The **event-driven** application subscription for this scenario is as follows (the XML representation of this subscription is depicted in Appendix E):

```

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

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

```

We have defined a route for John in which he passes inside the cinema located in the center of Enschede (named Alhambra). John walks in this route and when he

passes inside the Alhambra, an advertisement message is received in his terminal (Figure 41).

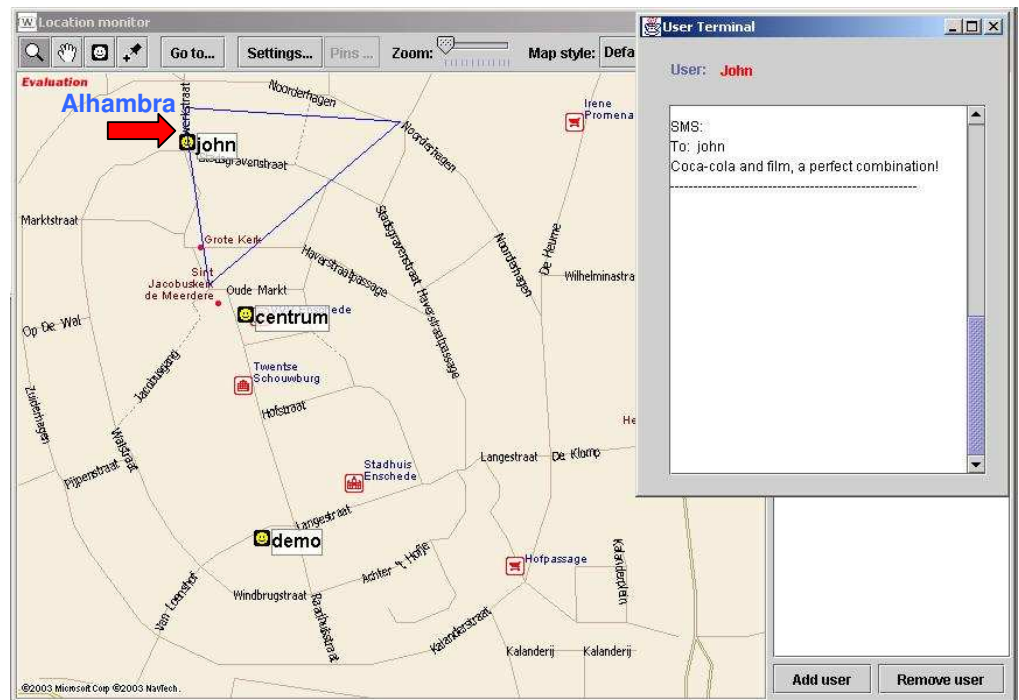


Figure 41 - Cinema scenario and John's terminal

The user terminal is implemented as a web service provider, i.e., every user terminal is supposed to provide at least a web service end point (WSDL description). In order to be able to callback users when needed (e.g., with the action **sendSMS**), the platform needs to have the address (URI) of the user's end-points. For instance, it is known by the platform that John's URI is **http://localhost:8080/axis/services/TerminalInterfaceServicePort**, which allows the platform to remotely call the **writeMsg** operation. This operation writes a text message on the user's terminal.

6.4.3 Proximity Scenario

In this simple proximity scenario, John is walking in the centre of Enschede and he receives a message in case Alice is physically located less than 15 meters far from him. The **event-driven** application subscription that represents this scenario is as follows (XML-format of this subscription is depicted in Appendix F):

```
ACTION
  SendSMS(entity.user.John,
    "Hey John, Alice is close by." )
GUARD
  CloseBy(entity.user.Alice, entity.user.John, 15)
```

Figure 42 depicts that John receives a message when he is passing close by Alice.

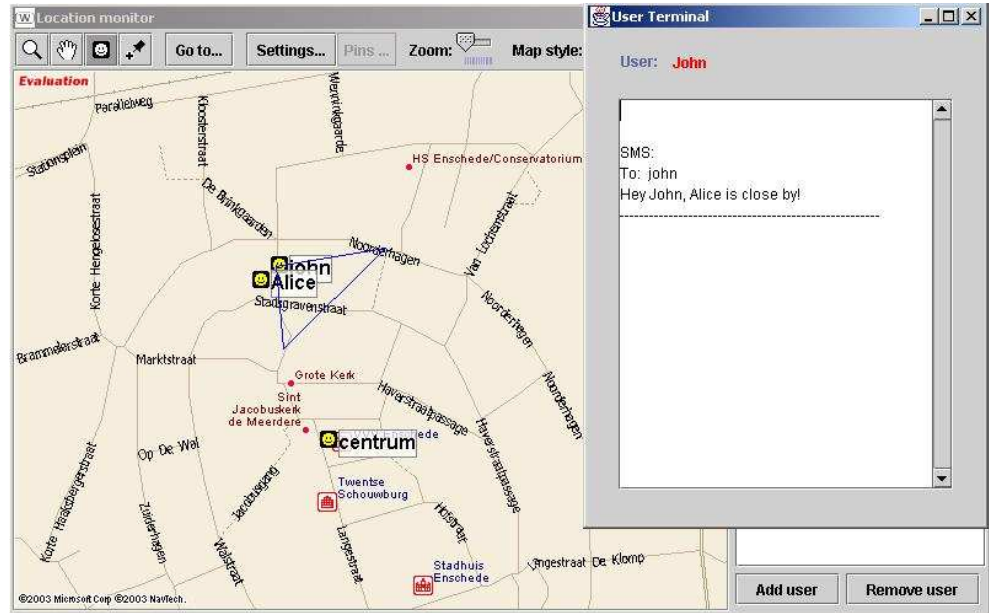


Figure 42 - Proximity Scenario and John's terminal

6.5 Concluding Remarks

Our prototype was able to demonstrate some of the concepts we have defined in the design of the platform. In particular, we have demonstrated the issues related to the *application-platform* interaction, such as dynamically deployment of applications during platform run-time through addition of application subscriptions. Moreover, the prototype includes the WSL parser, which is able to map application subscriptions written in XML-format to Java classes.

Further improvements and extensions of this prototype should address different types of contextual information, not just location. Moreover, it should implement a more elaborated *Context Interpreter*, which is capable of aggregating and inferring contexts (Section 5.2).

Finally, our prototype can be used as a starting point for integrating security and privacy, performance and scalability issues.

7 Conclusions

This chapter presents the main contributions of this thesis, draws some relevant conclusions and identifies points where further investigation is necessary.

This chapter is further structured as follows: Section 7.1 presents our general conclusions and summarizes the main contributions of this thesis, and Section 7.2 identifies some future work.

7.1 General Conclusions

We have proposed a high-level architecture of the WASP Platform, which is a services platform that supports mobile context-aware applications. Our efforts towards this architecture included (i) the study of the WASP project goals, (ii) a literature survey in context-aware computing, (iii) the identification of the essential challenges for building a services platform for context-aware applications, (iv) the design of the architecture based on the identified requirements and (v) the implementation of a prototype.

The proposed architecture tackles the following challenges: *Support for gathering, interpreting and storing of contextual information; Reactive behavior; Dynamic application deployment during the platform run-time; Coordination among different applications* and *Support for dynamic service discovery*.

Most of the approaches for building context-aware services platforms we have investigated do not explore the dynamic deployment of mobile context-aware applications on top of a services 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 run-time. 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, coined WASP Subscription Language (WSL), is used to specify how the platform must

react to a given correlation of events, potentially involving contextual information. A subscription in this language may specify different models of application - platform interaction, namely *request-response*, *time-driven* and *event-driven*.

We have used *Web Services* as a technology to enable the interactions of the platform with its environment. As a consequence, third party applications may 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.

For demonstration purposes, we have selected some of the proposed architectural elements to be prototyped. These elements are the *Subscription Manager*, the *Parser* and a simplified *Context Interpreter*. With respect to the WASP Subscription Language, we have defined an XML Schema that represents the WSL Syntax. Application subscriptions are written 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 platform run-time. We have illustrated the use of our prototype in several different application scenarios with different applications requirements.

Defining a complete architecture for a context-aware services platform is a non-trivial assignment. 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.

This thesis has identified basic architectural elements of the WASP platform, giving emphasis to the extensibility of the platform's generic functionality. A number of additional issues still have to be explored. Some are being investigated in parallel efforts and others are suggested for further investigation. Although the scope of this project does not comprise an exhaustive definition of all the identified architectural elements, the proposed architecture is a significant contribution, not only with respect to the problems it overcomes but also with respect to the elicitation of future work.

7.2 Future Work

The current WASP platform architecture does not explore all the identified challenging issues. The following list presents the topics which are indicated for further investigation:

- Exploration of *ontology-based* approaches to address *Context Modeling* issues. As discussed in Section 2.3.3, *ontologies* are believed to be a promising technology to model context. Parallel efforts inside the WASP project are investigating *ontology-based* approaches;
- The *Context Interpreter* architectural element could be further explored, especially with respect to the applicability of inference rules in order to capture higher level contextual information. As discussed in Section 5.3.2, inferring rules define logical relationships between information (context or not) in order to derive information that cannot be directly

sensed from the environment. This technique can greatly increase applications' context-awareness;

- Integration of *Security and Privacy* issues to the platform. As discussed in Section 4.3.1, the applicability of security and privacy concerns is essential for the success of a context-aware services platform. Parallel efforts inside the WASP project are being carried out in order to explore *Security and Privacy* concerns;
- Investigation of approaches to address a more integrated service discovery through the subscription language. As mentioned in Section 5.4.4, enhancements of our approach with respect to service discovery could be implemented by shielding application-UDDI+ interactions with the utilization of application subscriptions to specify UDDI+ services' requests;
- Addressing *scalability* and *performance* requirements. As mentioned in Section 4.4, *scalability* and *performance* issues are important crosscutting requirements for a services platform;
- Support for *Charging* requirements. As mentioned in Section 4.3.2, *Charging* requirements include the definition of a business model that defines requirements for the assignment of business responsibilities among the different parties involved in project. This is a relevant requirement for the success of a commercial services platform;
- Incorporation of solutions to the aforementioned issues in the prototype, for purposes of validation and demonstration.

References

- [1] Almeida, J., et al., Web Services Technologies. *WASP Deliverable: D3.1*, January 2003.
- [2] Banavar, G., et al., Challenges: An Application Model for Pervasive Computing. *Proceedings 6th Annual Intl. Conference on Mobile Computing and Networking (MobiCom 2000)*, Massachusetts, USA, August 2000.
- [4] Brown, P.J., et al., Context-Aware Applications: From the Laboratory to the Marketplace. *IEEE Personal Communications*, 4(5) (1997), pp. 58-64.
- [5] Buchholz, T., Context-aware Services for UMTS-Networks. *Summer School on Ubiquitous and Pervasive Computing*, Germany, August 2002.
[<http://www.inf.ethz.ch/vs/events/dag2002/program/ws/Buchholz.pdf>].
- [6] Chen, H., et al., An Ontology for Context-Aware Pervasive Computing Environments. *To appear in the Workshop on Ontologies and Distributed Systems (IJCAI 2003)*, Mexico, 2003.
- [7] Davies, N., et al., Developing a context-sensitive tour guide. *1st Workshop on Human Computer Interaction for Mobile Devices*, Scotland, 1998.
- [8] DeVaul, R. W., et al., The Ektara Architecture: The Right Framework for Context-Aware Wearable and Ubiquitous Computing Applications. *MIT Technical Report*, 2000.
- [9] Dey, K., et al., A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction Journal* 16, 24 (2001), pp. 97-166.
- [10] Dey, A. K., et al., Towards a Better Understanding of Context and Context-Awareness. *Technical Report 99-22*, Georgia Institute of Technology, 1999.
- [11] Domnitcheva, S., Location modelling: state of the art and challenges. *Distributed Systems Group Technical Report*, Department of Computer Science, ETH Zurich, Swiss Federal Institute of Technology.
- [12] Ebben, P., Blueprint and design of the WASP application platform. *WASP Deliverable: D2.2*, December 2002.
- [13] Esler, M., et al., Next Century Challenges: Data-Centric Networking for Invisible Computing. *Proceedings 5th Annual Intl. Conference on Mobile Computing Networking (MobiCom '99)*, August 1999.
- [14] Efstratiou C., et al., An Architecture for the Effective Support of Adaptive Context-Aware Applications. *Mobile Data Management 2001*, pp. 15-26.
- [15] Eustice, K., et al., A Universal Information Appliance. *IBM Systems Journal*, vol. 38 No. 4, 1999.

-
- [16] Fickas, S., et al., Software Organization for Dynamic and Adaptable Wearable Systems. *First International Symposium on Wearable Computers*, 1997, pp. 155-160.
- [17] Guha, R., Contexts: a formalization and some applications. *Technical Report*, Stanford University, Stanford, CA, 1992. [<http://www-formal.stanford.edu/guha/guha-thesis.ps>]
- [18] Halpin, T., Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design. Morgan Kaufman, Mountain View, CA, 2001.
- [19] Held, A., Modeling of context information for pervasive computing applications. *Proc. of the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI2002)*, Orlando, FL, Jul 2002.
- [20] Henderson, M. A *Framework for Event Correlation*. Master Thesis, October 1999.
- [21] Henricksen, K., et al., Generating Context Management Infrastructure from High-Level Context Models. *Proc. of the 4th International Conference on Mobile Data Management, Industrial Track Proceedings*. January 2003, Melbourne, Australia. pp. 1-6.
- [22] Henricksen, K., et al., Modeling Context Information in Pervasive Computing Systems. *Proc. of the First International Conference on Pervasive Computing, (Pervasive'2002)*, Zurich, August 2002.
- [23] Henricksen, K., et al., Infrastructure for Pervasive Computing: Challenges. *Proc. of the Informatik 2001: Workshop on Pervasive Computing*, Vienna, September 2001, pp. 214-222.
- [24] Indulska, J., et al., Experiences in Using CC/PP in Context-Aware Systems. *Proc. of the 4th International Conference on Mobile Data Management*, January, 2003, Melbourne, Australia, pp. 247-261.
- [25] Indulska, J., et al., An Open Architecture for Pervasive Systems. *Proc. of the 3rd International Working Conference on Distributed Applications and Interoperable Systems (DAIS 2001)*, Kraków, Poland, pp. 175-188.
- [26] Kindberg, T. et al. People, Places, Things: Web Presence for the Real World. *Hewlett-Packard Labs Technical Report HPL-2000-16*, 2000.
- [27] Laar, V., Requirements for the 3G Platform. *WASP Deliverable: D1.1*, January 2003.
- [28] Langheinrich, M., Privacy by Design – Principles of Privacy-Aware Ubiquitous Systems. *ACM UbiComp 2001*, Atlanta, GA, 2001.
- [29] Langheinrich, M., A privacy Awareness System for Ubiquitous Computing Environment. *UbiComp 2002*, Springer LNCS 2498, pp. 237-245.
- [30] Leiberman, H., et al., Out of Context: Systems That Adapt to, and Learn from, Context. *IBM Systems Journal* 39, 3&4 (2000), pp. 617-632.

- [31] Long, S., et al., Rapid Prototyping of Mobile Context-Aware Applications: The Cyberguide Case Study. *Proc. of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, November 1996.
- [32] McCarthy, J., Notes on formalizing context. *Proc. of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, Morgan Kaufmann, Mountain View, CA, 1993. [<http://www-formal.stanford.edu/jmc/home.html>].
- [33] Mansouri-Samani, M., *Monitoring of Distributed Systems*, Ph.D. Theses, Department of Computing, Imperial College, London, UK, 173 pages, December 1995.
- [34] Paolucci, M., et al., Importing Semantic web in UDDI. *Proc. of Web Services, E-business and Semantic Web Workshop*, 2002, pp. 225-236.
- [35] 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].
- [36] Pascoe, J., The Stick-e Note Architecture: Extending the Interface Beyond the User. *Proc. of the 2nd international conference on Intelligent user interfaces*, Florida, United States, January, 1997, p. 261-264.
- [37] Prokaev, S., et al., Context-Aware Services, *WASP Deliverable: D2.3*, December 2002.
- [38] Pokraev, S. et al., Extending UDDI with context-aware features based on semantic service description. *Proc. of the 1st Intl. Conf. on Web Services (ICWS 2003)*, Las Vegas, USA, June 2003.
- [39] Ryan, N.S., et al., Enhanced reality fieldwork: the context-aware archaeological assistant. *Computer Application in Archaeology, (CAA97), Digest of Papers*, British Archaeological Report Series, Archaeopress, Oxford, UK, 1997.
- [40] Schilit, B., et al., Disseminating Active Map Information to Mobile Hosts. *IEEE Networks*, 8(5) (1994), pp. 22-32.
- [41] Strang, T., et al., Service Interoperability on Context Level in Ubiquitous Computing Environments. *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet (SSGRR2003w)*, L'Aquila/Italy, January 2003.
- [42] Sun Microsystems, Java API for XML-Based RPC (JAX-RPC) Specification 1.0, JSR-101. [<http://java.sun.com/xml/downloads/jaxrpc.html#jaxrpcspec1>].
- [43] M. Theodorakis et al., Context in information bases. *Proc. of the 3rd International Conference on Cooperative Information Systems (IFCIS)*, New York, USA, August, 1998, pp. 260-270.
- [44] Want, R., et al., The Active Badge location system. *ACM Transactions on Information Systems* 10(1) (1992), pp. 91-102.

[45] Want, R., et al., The ParcTab Ubiquitous Computing Experiment. *Xerox Palo Alto Research Center Technical Report CSL-95-1*, 1995.

[46] WASP project [<http://www.freeband.nl/projecten/wasp/ENindex.html>].

[47] World Wide Web Consortium. Resource Description Framework (RDF): Concepts and Abstract Syntax. November, 2002. [<http://www.w3.org/TR/rdf-concepts/>].

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

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

Appendix A WSL - XML Schema

This appendix depicts the XML Schema representing the WSL syntax described in Section 5.5.1. It is used by applications and platform to syntactically check subscriptions written in XML format.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="subscription">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="scope" type="scopeType" minOccurs="0"/>
        <xs:element name="actions" type="actionsType"/>
        <xs:element name="guard" type="expressionType" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="scopeType">
    <xs:sequence>
      <xs:element name="collection" type="collectionExpressionType"/>
    </xs:sequence>
    <xs:attribute name="var" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="actionsType">
    <xs:sequence>
      <xs:element name="action" type="actionType" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="actionType">
    <xs:sequence>
      <xs:element name="param" type="expressionType" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="collectionExpressionType">
    <xs:choice>
      <xs:element name="select" type="selectType"/>
      <xs:element ref="entities"/>
      <xs:element name="function" type="functionType"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="expressionType">
    <xs:choice>
      <xs:element name="var" type="varType"/>
      <xs:element name="literal" type="literalType"/>
      <xs:element name="entity" type="entityType"/>
      <xs:element name="entity_attribute" type="entity_attributeType"/>
      <xs:element name="entity_context" type="entity_contextType"/>
      <xs:element ref="entities"/>
      <xs:element name="not" type="expressionType"/>
      <xs:element name="and" type="binopType"/>
      <xs:element name="or" type="binopType"/>
      <xs:element name="equal" type="binopType"/>
      <xs:element name="greaterthan" type="binopType"/>
      <xs:element name="lessthan" type="binopType"/>
      <xs:element name="select" type="selectType"/>
      <xs:element name="function" type="functionType"/>
    </xs:choice>
  </xs:complexType>

```

```
<xs:complexType name="binopType">
  <xs:sequence>
    <xs:element name="operand_a" type="expressionType"/>
    <xs:element name="operand_b" type="expressionType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="varType">
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="literalType">
  <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>

<xs:element name="entities" type="xs:string"/>

<xs:complexType name="entityType">
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="entity_attributeType">
  <xs:attribute name="entity_name" type="xs:string" use="optional"/>
  <xs:attribute name="entity_var" type="xs:string" use="optional"/>
  <xs:attribute name="attribute_name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="entity_contextType">
  <xs:attribute name="entity_name" type="xs:string" use="required"/>
  <xs:attribute name="context_name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="selectType">
  <xs:sequence>
    <xs:element name="collection" type="expressionType"/>
    <xs:element name="condition" type="expressionType"/>
  </xs:sequence>
  <xs:attribute name="var" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="functionType">
  <xs:sequence>
    <xs:element name="param" type="expressionType" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
</xs:schema>
```

Appendix B Subscription Interface - WSDL

This appendix depicts parts of the WSDL description for the platform interface that allows manipulation of subscriptions (addition, deletion and notification), i.e., that enables the application-platform interaction.

```
<?xml version="1.0" encoding="UTF-8"?>
<wSDL:definitions targetNamespace="http://wasp_platform"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:impl="http://wasp_platform-impl"
xmlns:intf="http://wasp_platform" xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"
xmlns:wSDLsoap="http://schemas.xmlsoap.org/wsdl/soap"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<types>
<schema targetNamespace="http://wasp_platform"
xmlns="http://www.w3.org/2001/XMLSchema">
<complexType name="SubscriptionDescription">
<sequence>
<element name="WslDescription" nillable="true" type="xsd:string"/>
<element name="XmlWslDescription" nillable="true" type="xsd:string"/>
</sequence>
</complexType>
<element name="SubscriptionDescription" nillable="true"
type="intf:SubscriptionDescription"/>
<complexType name="SubscriptionNotification">
<sequence>
<element name="notification_elements" nillable="true"
type="intf:ArrayOfSubscriptionNotificationElement"/>
<element name="subs_id" nillable="true" type="xsd:string"/>
</sequence>
</complexType>
<complexType name="SubscriptionNotificationElement">
<sequence>
<element name="text_element" nillable="true" type="xsd:string"/>
<element name="entity_element" nillable="true" type="intf:EntityElement"/>
</sequence>
</complexType>
<complexType name="EntityElement">
<complexContent>
<extension base="intf:SubscriptionNotificationElement">
<sequence>
<element name="entity_id" nillable="true" type="xsd:string"/>
<element name="entity_type" nillable="true" type="xsd:string"/>
<element name="attributes" nillable="true" type="intf:ArrayOfEntityAttribute"/>
<element name="contexts" nillable="true" type="intf:ArrayOfEntityContext"/>
</sequence>
</extension>
</complexContent>
</complexType>
<complexType name="EntityAttribute">
<sequence>
<element name="name" nillable="true" type="xsd:string"/>
<element name="value" nillable="true" type="xsd:string"/>
</sequence>
</complexType>
<complexType name="ArrayOfEntityAttribute">
<complexContent>
<restriction base="SOAP-ENC:Array">
<attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="intf:EntityAttribute[]"/>
</restriction>
</complexContent>
</complexType>

```

```

<complexType name="EntityContext">
  <sequence>
    <element name="name" nillable="true" type="xsd:string"/>
    <element name="value" nillable="true" type="xsd:string"/>
  </sequence>
</complexType>
<complexType name="ArrayOfEntityContext">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="intf:EntityContext[]"/>
    </restriction>
  </complexContent>
</complexType>
<complexType name="ArrayOfSubscriptionNotificationElement">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
wsdl:arrayType="intf:SubscriptionNotificationElement[]"/>
    </restriction>
  </complexContent>
</complexType>
<element name="SubscriptionNotification" nillable="true"
type="intf:SubscriptionNotification"/>
</schema>
</types>

<wsdl:message name="addSubscriptionCallbackResponse">
  <wsdl:part name="return" type="SOAP-ENC:string"/>
</wsdl:message>

<wsdl:message name="addSubscriptionResponse">
  <wsdl:part name="return" type="SOAP-ENC:string"/>
</wsdl:message>

<wsdl:message name="removeSubscriptionResponse">
</wsdl:message>

<wsdl:message name="addSubscriptionCallbackRequest">
  <wsdl:part name="in0" type="intf:SubscriptionDescription"/>
  <wsdl:part name="in1" type="SOAP-ENC:string"/>
</wsdl:message>

<wsdl:message name="removeSubscriptionRequest">
  <wsdl:part name="in0" type="SOAP-ENC:string"/>
</wsdl:message>

<wsdl:message name="addSubscriptionRequest">
  <wsdl:part name="in0" type="intf:SubscriptionDescription"/>
</wsdl:message>

<wsdl:message name="getNotificationRequest">
  <wsdl:part name="in0" type="SOAP-ENC:string"/>
</wsdl:message>

<wsdl:message name="getNotificationResponse">
  <wsdl:part name="return" type="intf:SubscriptionNotification"/>
</wsdl:message>

<wsdl:portType name="SubscriptionInterfaceServicePortType">

```

```
<wsdl:operation name="addSubscription" parameterOrder="in0">
  <wsdl:input message="intf:addSubscriptionRequest"/>
  <wsdl:output message="intf:addSubscriptionResponse"/>
</wsdl:operation>

<wsdl:operation name="addSubscriptionCallback" parameterOrder="in0 in1">
  <wsdl:input message="intf:addSubscriptionCallbackRequest"/>
  <wsdl:output message="intf:addSubscriptionCallbackResponse"/>
</wsdl:operation>

<wsdl:operation name="removeSubscription" parameterOrder="in0">
  <wsdl:input message="intf:removeSubscriptionRequest"/>
  <wsdl:output message="intf:removeSubscriptionResponse"/>
</wsdl:operation>

<wsdl:operation name="getNotification" parameterOrder="in0">
  <wsdl:input message="intf:getNotificationRequest"/>
  <wsdl:output message="intf:getNotificationResponse"/>
</wsdl:operation>
</wsdl:portType>

<wsdl:binding name="SubscriptionInterfaceServicePortSoapBinding"
type="intf:SubscriptionInterfaceServicePortType">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="addSubscription">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input>
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://wasp_platform" use="encoded"/>
    </wsdl:input>
    <wsdl:output>
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://wasp_platform" use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="addSubscriptionCallback">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input>
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://wasp_platform" use="encoded"/>
    </wsdl:input>
    <wsdl:output>
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://wasp_platform" use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="removeSubscription">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input>
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://wasp_platform" use="encoded"/>
    </wsdl:input>
    <wsdl:output>
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://wasp_platform" use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="getNotification">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input>
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://wasp_platform" use="encoded"/>
    </wsdl:input>
    <wsdl:output>
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://wasp_platform" use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
</wsdl:service>
```

```
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="removeSubscription">
  <wsdlsoap:operation soapAction=""/>
  <wsdl:input>
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="http://wasp_platform" use="encoded"/>
  </wsdl:input>
  <wsdl:output>
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="http://wasp_platform" use="encoded"/>
  </wsdl:output>
</wsdl:operation>
<wsdl:operation name="getNotification">
  <wsdlsoap:operation soapAction=""/>
  <wsdl:input>
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="http://wasp_platform" use="encoded"/>
  </wsdl:input>
  <wsdl:output>
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="http://wasp_platform" use="encoded"/>
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="SubscriptionInterfaceService">
  <wsdl:port binding="intf:SubscriptionInterfaceServicePortSoapBinding"
    name="SubscriptionInterfaceServicePort">
    <wsdlsoap:address
      location="http://localhost:8080/axis/services/SubscriptionInterfaceServicePort"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Appendix C User Location Service - WSDL

This appendix depicts parts of the WSDL description for the User Location Service which allows gathering users' locations in a Location Simulator.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://usercontextmanager.wasp.freeband.nl"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:apachsoap="http://xml.apache.org/xml-soap"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns1="http://common.wasp.freeband.nl"
xmlns:intf="http://usercontextmanager.wasp.freeband.nl"
xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"
xmlns:impl="http://usercontextmanager.wasp.freeband.nl"
xmlns="http://schemas.xmlsoap.org/wsdl/"><wsdl:types><schema
xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://common.wasp.freeband.nl"><import
namespace="http://schemas.xmlsoap.org/soap/encoding/"><complexType
name="LatLong"><sequence><element name="latitude" type="xsd:double"/><element
name="longitude" type="xsd:double"/></sequence></complexType></schema></wsdl:types>
  <wsdl:message name="setUserLocationTriggerRequest">
    <wsdl:part name="in0" type="xsd:string"/>
    <wsdl:part name="in1" type="tns1:LatLong"/>
    <wsdl:part name="in2" type="xsd:double"/>
    <wsdl:part name="in3" type="xsd:boolean"/>
    <wsdl:part name="in4" type="xsd:boolean"/>
    <wsdl:part name="in5" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="getUserLocationRequest">
    <wsdl:part name="in0" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="removeUserLocationTriggerRequest">
    <wsdl:part name="in0" type="xsd:int"/>
  </wsdl:message>
  <wsdl:message name="updateUserLocationTriggerResponse">
  </wsdl:message>
  <wsdl:message name="updateUserLocationTriggerRequest">
    <wsdl:part name="in0" type="xsd:int"/>
    <wsdl:part name="in1" type="tns1:LatLong"/>
    <wsdl:part name="in2" type="xsd:double"/>
    <wsdl:part name="in3" type="xsd:boolean"/>
    <wsdl:part name="in4" type="xsd:boolean"/>
    <wsdl:part name="in5" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="removeUserLocationTriggerResponse">
  </wsdl:message>
  <wsdl:message name="setUserLocationTriggerResponse">
    <wsdl:part name="setUserLocationTriggerReturn" type="xsd:int"/>
  </wsdl:message>
  <wsdl:message name="getUserLocationResponse">
    <wsdl:part name="getUserLocationReturn" type="tns1:LatLong"/>
  </wsdl:message>
  <wsdl:portType name="UserLocation">
    <wsdl:operation name="getUserLocation" parameterOrder="in0">
      <wsdl:input name="getUserLocationRequest"
message="intf:getUserLocationRequest"/>
      <wsdl:output name="getUserLocationResponse"
message="intf:getUserLocationResponse"/>
    </wsdl:operation>
```

```

        <wsdl:operation name="setUserLocationTrigger" parameterOrder="in0 in1 in2 in3
in4 in5">
        <wsdl:input name="setUserLocationTriggerRequest"
message="intf:setUserLocationTriggerRequest"/>
        <wsdl:output name="setUserLocationTriggerResponse"
message="intf:setUserLocationTriggerResponse"/>
        </wsdl:operation>
        <wsdl:operation name="updateUserLocationTrigger" parameterOrder="in0 in1 in2
in3 in4 in5">
        <wsdl:input name="updateUserLocationTriggerRequest"
message="intf:updateUserLocationTriggerRequest"/>
        <wsdl:output name="updateUserLocationTriggerResponse"
message="intf:updateUserLocationTriggerResponse"/>
        </wsdl:operation>
        <wsdl:operation name="removeUserLocationTrigger" parameterOrder="in0">
        <wsdl:input name="removeUserLocationTriggerRequest"
message="intf:removeUserLocationTriggerRequest"/>
        <wsdl:output name="removeUserLocationTriggerResponse"
message="intf:removeUserLocationTriggerResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="UserLocationServiceSoapBinding" type="intf:UserLocation">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getUserLocation">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getUserLocationRequest">
    <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://usercontextmanager.wasp.freeband.nl"/>
    </wsdl:input>
    <wsdl:output name="getUserLocationResponse">
    <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://usercontextmanager.wasp.freeband.nl"/>
    </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="setUserLocationTrigger">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="setUserLocationTriggerRequest">
    <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://usercontextmanager.wasp.freeband.nl"/>
    </wsdl:input>
    <wsdl:output name="setUserLocationTriggerResponse">
    <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://usercontextmanager.wasp.freeband.nl"/>
    </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="updateUserLocationTrigger">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="updateUserLocationTriggerRequest">
    <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://usercontextmanager.wasp.freeband.nl"/>
    </wsdl:input>
    <wsdl:output name="updateUserLocationTriggerResponse">
    <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://usercontextmanager.wasp.freeband.nl"/>
    </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="removeUserLocationTrigger">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="removeUserLocationTriggerRequest">
    <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://usercontextmanager.wasp.freeband.nl"/>
    </wsdl:input>
    <wsdl:output name="removeUserLocationTriggerResponse">
    <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://usercontextmanager.wasp.freeband.nl"/>
    </wsdl:output>
    </wsdl:operation>

```

```
</wsdl:binding>
  <wsdl:service name="UserLocationService">
    <wsdl:port name="UserLocationService"
binding="intf:UserLocationServiceSoapBinding">
      <wsdlsoap:address
location="http://client140.lab.telin.nl:8081/wasp/services/UserLocationService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Appendix D Policemen Scenario (WSL-XML)

This is the XML format of the Policemen Scenario's subscription.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Patricia Dockhorn Costa
(University of Twente) -->
<!--
Author: Patricia Dockhorn Costa (dockhorn@cs.utwente.nl)
Date: 19-06-2003

-->
<subscription xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="wsl.xsd">
<scope var="u">
  <collection>
    <select var="u2">
      <collection>
        <entities>entity.user.*</entities>
      </collection>
      <condition>
        <equal>
          <operand_a>
            <entity_attribute entity_var="u2" attribute_name="profession"/>
          </operand_a>
          <operand_b>
            <literal value="policemen"/>
          </operand_b>
        </equal>
      </condition>
    </select>
  </collection>
</scope>

<actions>
  <action name="notify">
    <param>
      <function name="list">
        <param>
          <var name="u"/>
        </param>
        <param>
          <select var="p">
            <collection>
              <entities>entity.user.*</entities>
            </collection>
            <condition>
              <and>
                <operand_a>
                  <function name="closeby">
                    <param>
                      <var name="p"/>
                    </param>
                    <param>
                      <var name="u"/>
                    </param>
                    <param>
                      <literal value="200"/>
                    </param>
                  </function>
                </operand_a>
                <operand_b>
```

```

                                <equal>
                                  <operand_a>
                                    <entity_attribute entity_var="p"
attribute_name="profession"/>
                                  </operand_a>
                                  <operand_b>
                                    <literal value="policemen"/>
                                  </operand_b>
                                </equal>
                              </operand_b>
                            </and>
                          </condition>
                        </select>
                      </param>
                    </function>
                  </param>
                </action>
              </actions>

            <guard>
              <function name="OnEvery">
                <param>
                  <literal value="10"/>
                </param>
              </function>
            </guard>

          </subscription>

```

Appendix E Advertisement Scenario (WSL-XML)

This is the XML format of the Cinema Scenario's subscription.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Patricia Dockhorn Costa
(University of Twente) -->

<subscription xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="wsl.xsd">
<actions>
  <action name="sendsms">
    <param>
      <entity name="entity.user.id1"/>
    </param>
    <param>
      <literal value="Coca-cola and film, a perfect combination!"/>
    </param>
  </action>
</actions>

<guard>
  <greaterthan>
    <operand_a>
      <function name="count">
        <param>
          <select var="c">
            <collection>
              <entities>entity.cinema.*</entities>
            </collection>
            <condition>
              <and>
                <operand_a>
                  <function name="inside">
                    <param>
                      <entity name="entity.user.id1"/>
                    </param>
                    <param>
                      <var name="c"/>
                    </param>
                  </function>
                </operand_a>
                <operand_b>
                  <equal>
                    <operand_a>
                      <entity_attribute entity_var="c" attribute_name="location"/>
                    </operand_a>
                    <operand_b>
                      <literal value="Enschede"/>
                    </operand_b>
                  </equal>
                </operand_b>
              </and>
            </condition>
          </select>
        </param>
      </function>
    </operand_a>
    <operand_b>
      <literal value="0"/>
    </operand_b>
  </greaterthan>
</guard>
</subscription>
```

Appendix F Proximity Scenario (WSL-XML)

This is the XML format of the Proximity Scenario's subscription.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Patricia Dockhorn Costa
(University of Twente) -->
<!--
Author: Patricia Dockhorn Costa (dockhorn@cs.utwente.nl)
Date: 01-06-2003
-->
<subscription xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="wsl.xsd">
  <actions>
    <action name="sendsms">
      <param>
        <entity name="entity.user.id1"/>
      </param>
      <param>
        <literal value="Hey John, Alice is close by!"/>
      </param>
    </action>
  </actions>
  <guard>
    <function name="closeby">
      <param>
        <entity name="entity.user.id1"/>
      </param>
      <param>
        <entity name="entity.user.id2"/>
      </param>
      <param>
        <literal value="15"/>
      </param>
    </function>
  </guard>
</subscription>
```