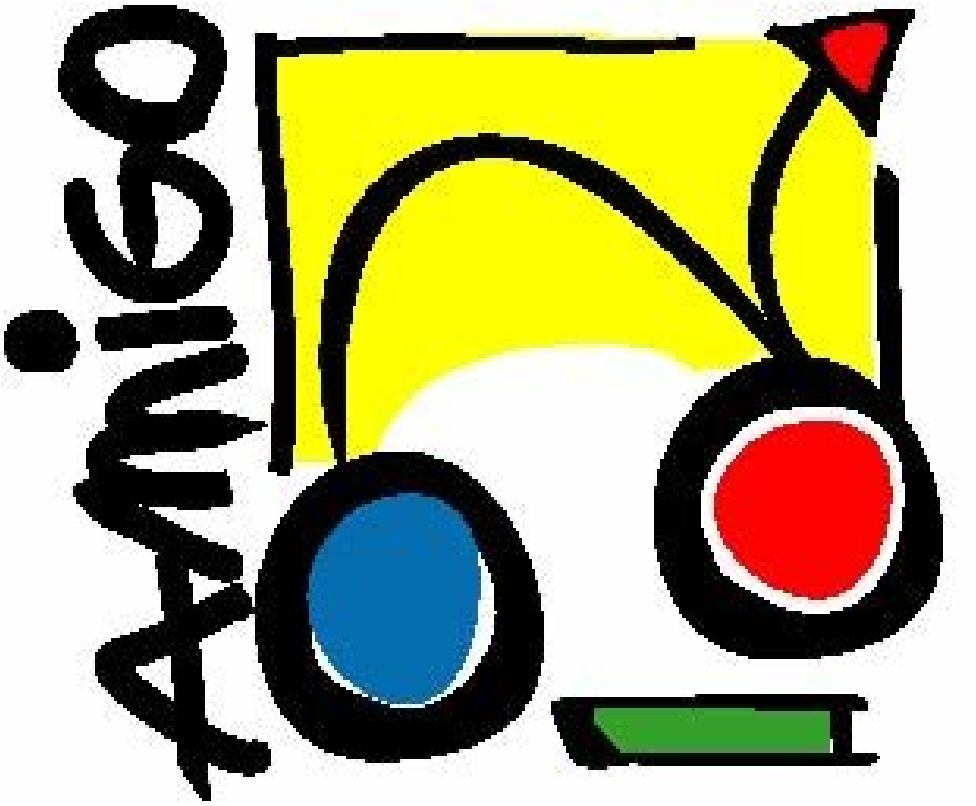


IST Amigo Project
Deliverable D4.7

Intelligent User Services
4 - Awareness and Notification Service
Software Developer's Guide

IST-2004-004182
Public

Project Number	:	IST-004182
Project Title	:	Amigo
Deliverable Type	:	Report

Deliverable Number	:	D4.7 (ANS contribution)
Title of Deliverable	:	4 - ANS Software developer's guide
Nature of Deliverable	:	Public
Internal Document Number	:	amigo_4_d4.7_final
Contractual Delivery Date	:	3- November 2007
Actual Delivery Date	:	14 January 2008
Contributing WPs	:	WP4
Author(s)	:	Thomas Dexheimer (Fraunhofer Institute SIT), Tom Broens (Telin, University Twente), Patricia Dockhorn Costa (Telin, University Twente), Luiz Olavo Bonino da Silva Santos (Telin, University Twente), Peter Vink (Philips)

Abstract

This document is the final programmers guide for the Awareness and Notification Service components. It describes how to install, use and edit the components of ANS.

Keyword list

ANS, Awareness, Notification, ECA-DL language.

Table of Contents

1 Introduction to ANS

1.1 Overview of ANS

The Awareness and Notification Service (ANS) provides the basic functionality required to develop applications allowing people and other applications to stay aware of any significant change in context with minimal effort. ANS is able to keep track of changes in various types of context, for example activities and presence of people.

ANS makes application layer services aware of context changes by notifying them. Applications register monitoring rules that specify what changes in context should be considered and when the changes occur, the client application is notified. From the user perspective, the Awareness and Notification Service provides notifications with appropriate rendering of intensity, based on the user's preferences and current context.

1.2 Benefit of using ANS

The Awareness and Notification Service enables to develop applications allowing users to stay aware of any significant change in context with minimal effort. From the system viewpoint ANS makes application layer services aware of context changes by notifying them. The benefit of using ANS is that applications do not have to care about subscribing to and monitoring of context data. These tasks are handled by ANS.

In order to be notified, applications have to register monitoring rules (the syntax of the rule language is specified in chapter 4.). Once an application has set its monitoring rules, ANS constantly checks them. If one rule evaluates to true, ANS notifies the application accordingly.

From the user perspective ANS provides notifications based on the user's preferences and their current context. In order to be notified appropriately, users create an individual user notification profile by using the User Modeling and Profiling Service (UMPS). The profile describes how and when a user wants to be notified. Before ANS sends a notification to an application, the service checks the notification profile of the user that is to be notified. Based on this profile, ANS sends a notification with an appropriate rendering of intensity. The intensity level ranges from 0 (notification at a later date) to 3 (alert), via 1 (ambient notification) and 2 (medium notification). The application receiving the ANS notification implements the notification of the user according to the intensity. For example using an "ambi-light" for an ambient notification.

1.3 Components

The Awareness and Notification Service consists of the following main components:

- **Rule manager**
  Handles the input of notification rules and contains the *IManageRules* interface.
- **Controller**.
  Handles the utilization of notification rules and contains the rule engine.
- **Event Monitor**
  handles the interface towards Context Management Service.
- **NotificationProfilemanager**
  Handles User preferences towards notification and interface towards User Profiling and Modeling Service (UMPS).

- **Notifier**
  Handles the output to User/Services according to the demanded level of intensity.

Only the combined ANS stack is offered as a service within Amigo. It is not the intention to offer the components as separate services. The design of ANS was presented in [Ami06b]. More information on each component is provided in the following sections.

1.3.1 Rule Manager

Provider

TELIN

Introduction

From an external perspective, the Rule Manager enables client users/application developers to introduce and manage their ANS rules in the system. Internally, the Rule Manager parses these rules and checks them for syntax and semantic correctness. The rules are conditioned and then forwarded to be processed by the Controller. The condition part of the rule is de-integrated into atomic event requirements passed to the Event Monitor.

Development status

Methods to subscribe, start, stop and unsubscribe rules are ready.

Indented audience

Client application developers and end-users of the system. Although, a user interface has to be used for efficient rule creation by end-users.

License

Apache v2 license

Language

Java

Environment (set-up) info

Hardware: PC

Software: J2SE

Platform

Computer system with a JVM needed to execute the java binaries.

Tools

Eclipse for compilation of the source.

Documents

See D4.2 for a detailed overview of the design of the RuleManager.

Tasks

-

Bugs

-

Patches

-

1.3.2 Controller

Provider

TELIN

Introduction

After subscribing a client's rule, the Controller receives event notifications from the Event Monitor. The Controller checks the event notifications against the subscribed rules and when the context changes defined in a rule have been detected, invokes the Notifier to proceed to client application's notification.

Inside the Controller there is a parser and a rule engine. The parser translates the Event Condition Action (ECA) rule received to a version compatible with the used rule engine. The rule engine is responsible for receiving rules and facts as inputs and deriving conclusions as output. It contains an inference engine, a rule base and a working memory. Currently, Amigo uses the JESS engine in ANS. However, the Controller design was kept intentionally generic enough to accommodate other rule engine as well.

Development status

The *upon*, *when, do* and *lifetime* clauses of the ECA rule language have been implemented as well as basic relations such as *isLocatedIn*, *hasHeartRate*, *hasWeight*, *hasSchedule* and *hasBloodPressure*. The current version is completely integrated with the Event Monitor and the Notifier. Implementation of further concepts of the ECA rule language is still required although the current implementation supports the defined usage scenarios.

Indented audience

The controller is an internal component of ANS. However, the architecture with a rule engine and a parser can be used by other project partners as-is or as a reference model.

License

Apache v2 license

Language

Java

Environment (set-up) info

Hardware: PC

Software: J2SE

Platform

Computer system with a JVM needed to execute the java binaries.

Tools

Eclipse for compilation of the source.

Documents

See D4.2 for a detailed overview of the design of the Controller.

Tasks

Implement the other concepts of the ECA rule language. Store the subscribed rules using a persistence mechanism.

Bugs

-

Patches

-

1.3.3 Event Monitor

Provider

Philips

Introduction

The main functionality of the Event Monitor component is to allow easy access to Context Management Service (CMS) components and to manage the combinations of events and context sources.

Development status

Updated version available, supports a limited number of context types, simple refresh mechanism implemented, supports latest CMS interface (including WS eventing).

Indented audience

Component internal to ANS, but intended for programmers.

License

Philips license, BSD style

Language

Java

Environment (set-up) info

Event monitor is an integral part of ANS, it uses the same set-up.

Tasks

Supports only limited number of concepts from ontology.

Patches

-

1.3.4 Notification Profile Manager

Provider

Fraunhofer IPSI

Introduction

The Notification Profile Manager enables ANS to access the notification preferences of users that are externally stored in the UMPS. This component also makes sure that the settings of a retrieved user notification profile are not conflicting.

Development status

The component has been implemented and is ready for being tested by users. Further implementation depends on User Profiling And Modeling Service (UMPS).

Indented audience

Project partners at first and then later also other developers of custom AMIGO services and applications.

License

LGPL

Language

Java

Environment (set-up) info

Hardware: PC

Software: J2SE and Jess rule engine

Platform

Computer system with a JVM needed to execute the java binaries.

Tools

Eclipse to compile the source code.

Documents

-

Tasks

-

Bugs

-

Patches

-

1.3.5 Notifier

Provider

Fraunhofer IPSI

Introduction

The main task of the Notifier is to determine the right intensity for notifications and to send them to the right applications.

Development status

The basic functionality of the component has been implemented. Further implementation will provide more flexibility regarding the way of notifying users.

Indented audience

Project partners at first and then later also other developers of custom AMIGO services and applications.

License

LGPL

Language

Java

Environment (set-up) info

Hardware: PC

Software: J2SE and Jess rule engine

Amigo IST-2004-004182 73/85 March 2006 Public

Platform

Computer system with a JVM needed to execute the java binaries.

Tools

Eclipse to compile the source code.

Documents

-

Tasks

-

Bugs

-

Patches

-

2 How to set up ANS

This chapter lists all resources that will be needed by a developer setting up ANS. It is explained where to get them and how to set them up. Please regard that this description is targeting for windows environments only. Setting up ANS within different environments will be similar to the steps described below but may differ in some points.

1.4 System requirements

ANS has been implemented to run in an Open Service Gateway Initiative (OSGI) framework. The open-source OSGI implementation Oscar has been used. Since Oscar is a java-based implementation, you need to have the Java Runtime Environment installed on your system (http://java.sun.com/j2se).

For further information about Oscar and OSGI, see http://oscar.objectweb.org/.

1.5 Download and Installation

ANS has three dependencies

- OSCAR platform (either linux or windows) and the (OSGI) components to run this platform (see above).
- Amigo OSGi-based framework (amigo_core, amigo_ksoap, amigo_wsdiscovery).
- CMS framework, consisting of at least one broker and context sources, context manager and context helper. Context broker in its turn depends on jena.
    - Broker can be installed on the same OSCAR platform, but this is not required, refer to CMS part of D4.6 which explains how to set up CMS.

Besides the core libraries of ANS, the following components are provided:

- Example client application
- Example context source

Please note: A special stand-alone version has been created for the tutorial which does not need CMS.

1.5.1 Oscar

- First you need to download the Oscar OSGI implementation. You can get it from http://amigo.gforge.inria.fr/obr/tools/index.html.
- After you have downloaded the Oscar file, unzip it.
- Place the ECAXML.xsd file in the Oscar root directory. It is required to start the rules you subscribe.
- Start Oscar by opening oscar.bat.
- Type in a name for your profile, and you will see the GUI.
- OSCAR will have installed the OSGI core bundles

1.5.2 Amigo core bundles

- These can be found at http://amigo.gforge.inria.fr/obr/v2/repository.xml

- These bundles will be installed through dependencies in ANS if this does not work you can do the following
    - Go to the OBR window of OSCAR, and type the address above
    - Install the following bundles: amigo_ksoap_binding, amigo_ksoap_export, amigo_core, amigo_wsdiscovery

1.5.3 CMS bundles

- These can be found at http://amigo.gforge.inria.fr/obr/v2/repository.xml
- The broker and manager are needed, but do not need to run in the same environment. In this example however, we have included them in order to keep one running overview.
- The helper bundles will be installed through dependencies in ANS if this does not work you can do the following
    - Go to the OBR window of OSCAR, and type the address above
    - Install the following bundles: context_broker, context_manager, context_helper (these will in their turn depending on jena)
    - It is possible to run the broker and the manager also on a different platform (they are accessed through discovery mechanism)

1.5.4 ANS Bundles

These can be found at http://amigo.gforge.inria.fr/obr/v2/repository.xml

ANS exists of three parts:

- A rule engine, in this case a copy of JESS is provided. Users may select any other rule engine, but then have to implement a new wrapper analog to the wrapper for the JESS engine (amigo_ans_jess.jar)
- ANS API is exported as a separated package ans_api.jar
- The ANS core service (amigo_ans.jar)

All parts have been delivered as jar files and can be executed in the amigo framework/Oscar environment.

In addition, users can download:

- an example client application
- an example context source

ANS Profile

OSCAR allows storing a collection of jars in a profile, typically, an ANS profile contains ANS and all its dependencies, though users can decide to run CMS components in a separate environment.

The figure below shows an example of an ANS profile

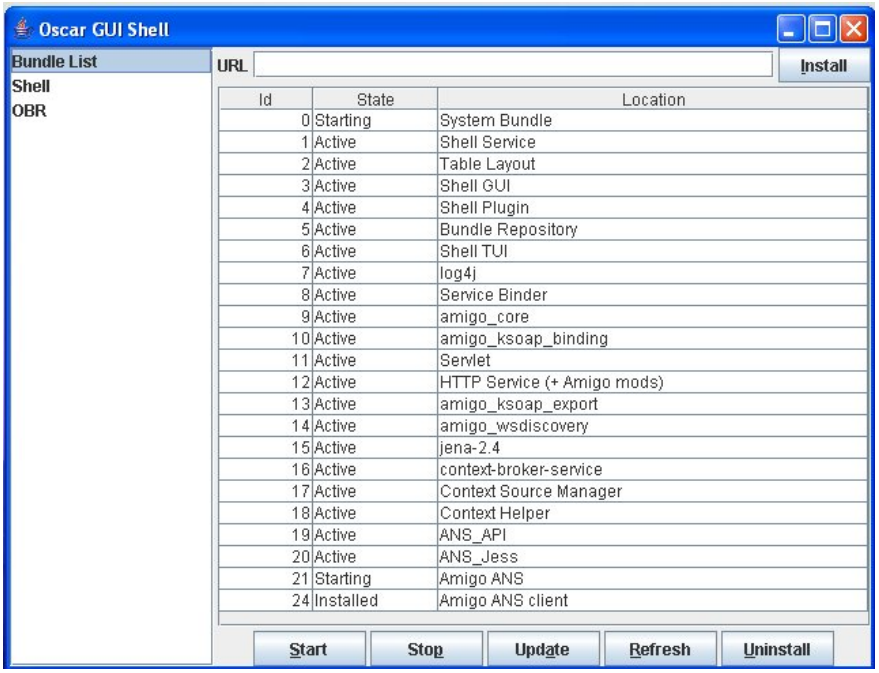


Figure: 1: ANS Profile

1.5.5 Configure

ANS requires the file ECAXML.xsd to be present in the working directory; this file can be found in the root directory.

1.5.6 Compile

The ANS source code can be found at:

[Amigo gforge repository]/ius/awareness_notification/…

Releases can be found in 'branches/' directory. The latest unreleased version is in the 'trunk/' directory.

ANS contains 3 subprojects:

- ans (ans core components, containing main components)
- ans_client (example ans client)
- ans_api (the API for managing communication between ANS and clients)

Using eclipse [.project] you can compile a new version of ANS, typically this will create a new version of ANS core (ans.jar) and unchanged copies of the rule engine and API jars.

The ANS project has been set up so that it can be debugged from within Eclipse while running in an Oscar environment. For this a special debug configuration file can be created in Eclipse as follows:

- use org.ungoverned.oscar.main as main class
- add oscar libraries (osgi.jar moduleloader.jar, oscar.jar) to classpath

- use Oscar directory as working directory
- (Oscar requires a console for input/output)
- Create a profile containing the libraries as shown above

1.5.7 Tools/Examples

ANS Client example

- To install the ANS Client, start another instance of Oscar.

  Important Note: If you start both instances on the same system, you will have to change the pre-defined configuration of one instance because otherwise service and client would listen on the same port.

  To do this, make a copy of bundle.properties in the lib directory, change the port number in the last line into a port number of your choice and save it under a different name. Then make a new copy of oscar.bat/oscar.sh, save it under a different name and add the content

  "-Doscar.bundle.properties=lib/<filename>"

  after the passage "-Dlog4j.configuration=file:lib/log4jbasic.properties".

  Then start the second instance with this batch/script file and enter a new profile name.
- Install the same bundles as for ANS, except "Amigo Client" instead of "Amigo ANS".
- After you have done this, you'll see the Client's GUI and the ANS and ANS Client are ready for use as described in the following chapter.

There is no ANS client provided for C#, though Appendix C shows how such a client could be constructed.

ANS Test context source

A simple context source is provided that can be used together with ANS to provide context input. It is explained in Appendix C.

3 Concepts and Interface

This chapter explains how to use ANS. It shows the interface and explains the concepts related to the rules that are shared on this interface.

1.6 How to find ANS

ANS can be found by looking for a service with an Uniform Resource Name (URN) equal to “amigo” and type equal to “ans“. The Amigo discovery framework will ensure that all available URLs are returned (users are free to choose any framework). Typically only one instance of ANS will be running in an amigo network, but if there are more, then anyone will do.

1.7 ANS Interface

ANS exports two Interfaces:

- I_manageRules, is used by clients for rule management.
- I_ReceiveContext, is a call back mechanism where context sources from CMS can provide information. Though this interface still exists, it is encouraged to use WS eventing instead.

A client to ANS should implement the following interface:

- I_notify, which is the function called by ANS (the actual notification).

1.7.1 Managing rules

After the initial step of retrieving the ANS service, the application is able to manage ECA-DL rules. Functionality that is offered by ANS is:

- Creating, updating and deletion of rules;
- Querying of rules;
- Enabling and disabling of rules;

This functionality is expressed in the IManageRule interface, which is exported to the application that wants to use ANS.

```
public interface IManageRule {
      public int subscribe(Rule rule, String applicationID) throws
         RuleFormatException;
      public boolean unsubscribe(int ruleID);
      public boolean updateRule(int ruleID, Rule rule);

      public Rule queryRule(int ruleID);

      public boolean startRule(int ruleID);
      public boolean stopRule(int ruleID);
}
```

The first three methods express creation, deletion and updating of rules. The subscribe method takes a rule expressed in the ECA-DL XML format. Additionally, it takes an application ID which uniquely specifies the application that creates the rule. The method returns the rule identification. This rule identification uniquely specifies the rule and can be used to delete or

update the rule. If the subscription fails (e.g. rule expressed in malformed ECA-DL syntax) a RuleFormatException is thrown. The queryRule method can be used to retrieve the ECA-DL XML of a rule with a specific rule ID. The last two methods can be used to start or stop the evaluation of a rule. When a rule is stopped it is not removed from the system but merely ignored by the evaluation engine. To remove a rule the unsubscribe method has to be invoked.

Example

Below we give a typical code section of an application that manages rules using ANS. We abstract from the Amigo middleware specific code.

```
// We assume the ANS service is found.
IManageRule rulemanager;

// Example rule in
Rule myRule = [see examples in section 3.4.2];
Rule myRule2 = [see examples in section 3.4.2];

// Add a rule to the system
Try{
   int myRuleID = rulemanager.subsribe(myRule, "ANS-consumer-1");
}catch(RuleFormatException ex){
   System.out.println("Error subscribing rule");
}

// Update myRule with myRule2
boolean update_status = rulemanager.updateRule(myRuleID, myRule2);

// Delete rule
Boolean delete_status = rulemanager.unsubscribe(myRuleID);

// Query a rule
Rule myRule3 = rulemanager.queryRule(myRuleID);

// Start and Stop a rule
rulemanager.startRule(myRuleID);
rulemanager.stopRule(myRuleID);
```

In section 3.3, the rule syntax is described.

1.7.2 Receiving a notification

Clients should implement the following interface:

«interface»**amigo.ans.api.INotifyApplication**
+notify(in message, in userID, in intensity)

This function will be called by ANS when a notification is triggered. The value will be the required intensity.

1.7.3 Providing context

Providing context follows the interfaces as it has been described in CMS (see D4.6, CMS contribution). In addition, context sources are free to choose between the WS eventing mechanism and callback function for providing their data.

1.7.4 Using deployment framework

Developers are free to select a development framework. Amigo provides two deployment frameworks that handle issues such as discovery and security and thereby reducing development effort:

- For the OSGI based framework an example client has been written. This client is also used in the chapter 4 “Tutorial”.
- For the .net based framework, the way of using ANS is identical. A small example application is provided in Appendix B.

1.8 ANS Concepts

1.8.1 ECA-DL Rule Language

ECA-DL is a domain specific language developed with the purpose of specifying event-condition-action (ECA) rules to be used in context-aware scenarios. It is the foundation of the Awareness and Notification Service (ANS), where it enables application developers to specify context-aware reactive behaviors in a flexible way.

Rules in ECA-DL are composed by three parts: *Event*, *Condition* and *Action*. At first, the *Event* part models an occurrence of interest in the context. The *Condition* part specifies a condition that must hold prior the execution of the action. At last the *Action* part defines which methods to invoke on notification.

ECA-DL is defined upon two complementary foundations: information and behavior foundations. *Information foundation* refers to the representation of the applications’ universe of discourse, i.e., the Amigo ontologies. For example, we should be able to express within ECA rules whether persons are in the house or not, whether objects are plugged in or not, whether persons and objects are collocated, etc. *Behavior foundation* of the ECA language refers to the dynamics of rule execution, i.e. how and when a rule should be executed and what are the elements of the language that should be used to perform a particular piece of reactive behavior.

For the information part, the Amigo ontology should be referenced. ANS assumes that one is only allowed to use a piece of knowledge in the ECA rule, if this has been previously defined in

the ontology. If the ontology does not define the concept co-location, for example, this concept cannot be referenced in ECA rules.

This tutorial concentrates on the behavior part of the language, and indicates how to make references to concepts of the ontology. It is beyond the scope of this document to discuss ontology definitions. For more information on the Amigo ontology see [Ami06a]. The remainder of this section discusses the three parts of an ECA-DL (event-condition-action), the syntax of the language using XML and gives some examples of ECA-DL rules.

1.8.2 Event

An event expresses a change in state, which is of interest to particular applications or users. For example, an application may be interested to know when Jerry enters the TV room, when Jerry and Roberto get close, or when Pablo becomes online in MSN. These scenarios refer to changes of state:

- Jerry *enters* TV room: State (Jerry is not in TV room) followed by State (Jerry is TV room);
- Jerry and Roberto *get* close to each other: State (Jerry and Roberto are far from each other) followed by State (Jerry and Roberto are close to each other);
- Pablo *becomes* online: State (Pablo is offline) followed by State (Pablo is online).

In ECA-DL we allow the definition of events in two ways: expressed as an *explicit* change of state, or expressed as an *event description*. Consider the examples defined above. We may express the situation "Jerry *enters* room" as EnterTrue (LocatedIn (Jerry, TVRoom)) or EnterRoom (Jerry, TVRoom). Similarly, we may define "Jerry and Roberto *get* close" as EnterTrue (CloseBy (Jerry, Roberto)) or GetClose (Jerry, Roberto)). Finally, we may define "Pablo *becomes* online" as EnterTrue (OnLine(Pablo)) or BecomesOnline(Pablo).

When defining in the ECA rule the change of state, for example using the state transitions EnterTrue and EnterFalse, the application developer expresses events by explicitly defining the state transition for a given situation. Conversely, application developers may express events by using pre-defined event descriptions. This requires these events to be previously defined (in the amigo ontology).

For using explicit state transitions, we have defined the following. There are three possible states (true, false and unknown) and six state transitions. The unknown state accommodates uncertainty of context information (when the value of context information in unknown). **Error! Reference source not found.** presents the state transitions:

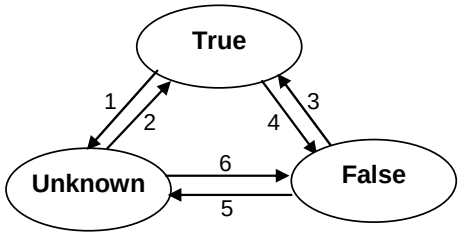


Figure: 2 State Transitions for a situation

Events can be any of the following transitions, for a given situation S:

```
EnterTrue(S) – transition 2 or 3
EnterFalse(S) – transition 4 or 6
EnterUnknown(S) – transition 1 or 5
ExitTrue(S) – transition 1 or 4
```

```
ExitFalse(S) – transition 3 or 5
ExitUnknown(S) – transition 2 or 6
TrueToFalse(S) – transition 4
TrueToUnknown(S) - transition 1
FalseToTrue(S) – transition 3
FalseToUnknown(S) – transition 5
UnknownToTrue(S) – transition 2
UnknownToFalse(S) – transition 6
Changed (S) – any transition
```

Currently, ANS implements EnterTrue and EnterFalse (used only with the IsLocatedIn relation) transitions. In summary, application developers can express events in the following ways:

- StateTransition (Situation); where situation defines a State (parameters); or,
- EventDescription (parameters);

1.8.3 Condition

The condition part of a rule describes extra conditions that must hold prior the invocation of a notification. It differs from the event part, since it does not define a change of state it only specifies additional requirements (states) that must hold. In addition to specifying an event, application developers may be interested in more specific situations. For example, an application may need to be notified upon Jerry entering the TV room, with the condition that Jerry should be alone. The condition that Jerry should be alone does not define a state transition; it just expresses extra requirements for the notification to be invoked.

Conditions are logical expressions that inquire whether (a combination of) situations are true or false. Situations refer to concepts defined in the Amigo ontologism.

1.8.4 Action

Actions describe operations that should be invoked when both the event and conditions parts of rules are fulfilled. In ANS, actions are restricted to notification operations that may send notification messages to end-users or application components. The way notifications are delivered depends on the users' preferences and context, as shown in the tutorial (Chapter 4)

1.9 ECA-DL Syntax

Based on the issues discussed in the previous sections, we have identified the basic requirements for ECA-DL with respect to elements in the language: (i) a way to specify (correlation of) events, (ii) a way to specify conditions, (iii) a way to specify notifications. These requirements resulted in the clauses **Upon**, **When** and **Do**, respectively. Events are defined in the **Upon** clause, while conditions are specified in the **When** clause and finally, notifications are specified in the **Do** clause. In case there are no conditions to be specified, the **When** clause may be omitted.

ECA rules can be either parameterized or not. Parameterization is necessary when the rule should be applied to a collection of entities. It would be cumbersome to write a rule for each target entity. For example, a medical clinic would like to apply a general rule (notify when sugar levels go above 110) to all patients suffering from diabetes. Parameterization allows the specification of a single rule to be executed for all the diabetic patients. We have introduced the **Scope** clause to define rule parameterization.

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 context and/or attributes constraints. For example, it may be necessary to select all users that are in the house and taking a shower, or we would like to select all devices that are currently being

used, or even all the patients of the clinics that have diabetes. Currently, the clauses Scope and Select are not supported by the ANS implementation.

Each rule is associated with a lifetime, which can be **always**, **once**, **from <start> to <end>**, **to <end>**, **<n> times**, **frequency <n> times per <period>**. Always defines that a rule should be triggered whenever events and conditions turn true; once defines that a rule should be triggered one time, and then should be deactivated; from <start> to <end> defines a period for the rule to be active for triggering; to <end> defines when a rule should be deactivated; <n> times says the number of times a rule should be triggers; frequency <n> times per <period> defines the number of times a rule should be triggered in a certain period of time, for example, once a day or twice a week. Currently, the lifetime clause can be set using the always or once parameters.

The non-parameterized rules are composed by the basic structure:

Upon <event-expression>

When <condition-expression>

Do <notifications>

<lifetime>

Table 1 gives an overview of the clauses with their main objectives. More details are given in the following paragraphs.

Table 1: Clauses of ECA-DL

Clause	Semantics
Select	Allows the selection from a collection using filtering expressions (logical expressions)
Upon-When-Do	Allows specification of an action (**Do**) which is triggered upon the occurrence of an event (**Upon**) respecting general conditions defined in the **When** clause
Scope	Allows parameterization of ECA rules

The Extended Backus Naur Form (EBNF) syntax of the language is depicted in Figure 4. While this concrete syntax is provided here, alternative representations could also be defined. For example, XML schemas (Appendix A) and UML metamodels (Figure 3).

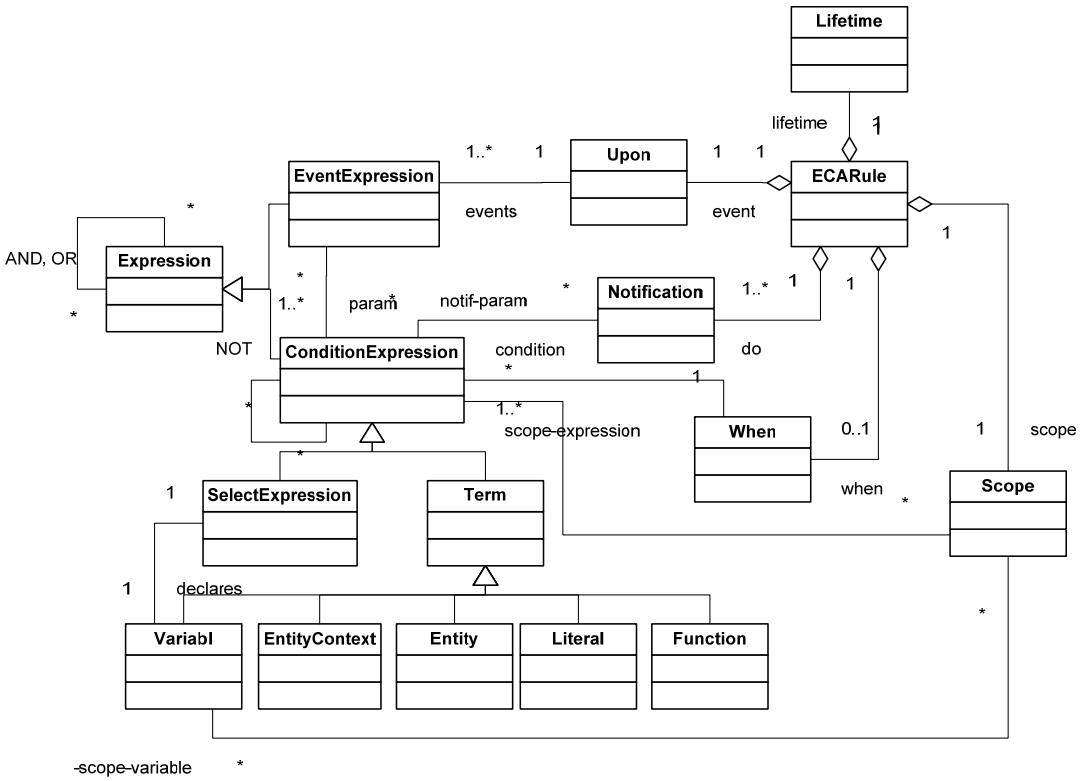


Figure: 3: ECA-Dl syntax in UML representation

The following paragraphs detail the EBNF syntax depicted in **Error! Reference source not found.**.

- A *notification* has a name and possibly zero or more parameters, which are defined as *expressions*.
- 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 followed by a filtering expression. For example, Select (user.*; u; isDriving(u)) 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* defines an object of interest in the system. For example, users, devices, buildings, rooms and spaces are *entities*. The syntax defines that an entity is specified with an entity type and a unique identification of the entity (entitytype.id). For example, if the unique Id of user Jerry is Jerry@JGAV3006.amigo.net, in ECA-DL this would be (user.Jerry@JGAV3006.amigo.net).
- The word “entities” stands for a collection of all instances of a given entity type. For example, user.* is the collection of all users registered in the system. If a more specific (filtered) collection is needed, it has to be explicitly defined 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 user.Jerry.location, which is referring to the location (context) of a specific user (John)

or it can be 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 user.John.address, which is referring to the address (attribute) of a specific user (John) or it can be user.*.address, which is referring to the address of all users in the collection.
- Syntactically similar to a *notification*, a *function* has a name and possibly zero or more parameters, which are defined as expressions. *Functions* may have any returning value type. However, typically, functions return True or False. Examples of functions are isDriving (John) and isInRoom (Mary, TVRoom).
- 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 symbol "_" (underscore).
- The *unop* (unary) operator is the NOT logical operator.
- The *binop* (binary) operators are he AND and the OR logical operators and the comparison operators > (greater than), < (less than), == (equal to), <= (greater than or equal to), <= (less than or equal to).
- The *event expression* specifies the possible (combination of) event (s): *event description*; *event transition* or a binary event operation over two event expressions.
- An *event description* defines an event, which has a name and possibly zero or more parameters. For example, EnterRoom (Maria, TVRoom) and EnterHouse (Jerry, House);
- An *event transition* explicitly defines the transition of states depicted in **Error! Reference source not found.**. Examples are EnterTrue (isInRoom (Mary, TVRoom)) EnterFalse (isDriving(John)).
- An *event binop operator* can either be an AND or OR logical operators.

```
ECARule          :   ruleparam | rulenotification
ruleparam        :   scope "{" rulenotification "}"
rulenotification :   "Upon" expr ["When" expr]"Do"
                      notifications ident
scope            :   "Scope (" expr "; " var ")"
notifications    :   notification ";" [notifications]
notification     :   ident "(" [expr ( ","
                     expr)*] ")"
expr             :   term | unop expr | expr binop
                     expr | select | "(" expr ")"
eventexpr        :   eventdesc | stateevent |
                     eventexpr eventbinop eventexpr
select           :   "Select (" expr ";" var ";"
                     expr ")"
term             :   entitycontext |
                     entityattribute | entities |
                     entity | literal | function |
                     var
entities         :   ident ".*"
unop             :   "NOT"
binop            :   "AND" | "OR" |  ">" | "<" |
                     "==" | "<=" | ">="
eventbinop       :   "AND" | "OR"
entitycontext    :   entity "." context | entities
                     "." context
entityattribute  :   entity "." attribute |
                     entities "." attribute
function         :   ident "(" [expr ( ","
                     expr)*] ")"
eventdesc        :   ident "(" [expr ( ","
                     expr)*] ")"
stateevent       :   ident "("ident "(" [expr ( ","
                     expr)*] "))"
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: 4: EBNF syntax for ECA-DL

1.9.1.1 Semantics of the Clauses

The following of paragraphs present the semantics of the ECA-DL 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 (user.*, u; u.location.city == “Enschede”), which selects all the users located in Enschede. Alternatively, we could define Select (user.*, u; isLocatedIn (u, “Enschede”)).

The Upon clause

The Upon clause defines the events (or combination of events) that should happen to enable the notification to be triggered. Its abstract syntax is:

```
Upon <correlation-of-events>
```

A concrete example is:

Upon EnterTrue(isInRoom (Maria, TVRoom) AND isInRoom (Jerry, TVRoom)), which generates an event when both Maria and Jerry are in the TV room.

The When clause

The When clause defines conditions that should hold upon occurrence of events specified in the Upon clause, in order to enable the triggering of notifications. This clause is optional, and can be omitted in case no extra conditions to events are necessary. Its abstract syntax is:

```
When <correlation-of-conditions>
```

A concrete example is:

When isOnline (computer) AND isOwned (computer, Jerry)

The Do clause

The Do clause specifies the notifications that have to be sent whenever the Upon and the When clauses are fulfilled. Its abstract syntax is:

```
Do <notifications>
```

A concrete example is:

Do notify (Jerry, “Maria is in the house”); notify (Maria, “Jerry is in the house”)

The Scope clause

The Scope clause defines a collection of target entities for which the ECA rule should be applied. The Upon-When-Do is nested in the Scope clause. The scope clause has the following abstract syntax:

```
Scope (<collection-of-entities>; var)
{ Upon <correlation-of-events>
  When <correlation-of-conditions>
```

```
  Do <notification>
}
```

Consider the following scenario:

For all family members (Smith family), when they enter home (SmithHome), notify other family members, who are located in the house. In ECA-DL:

```
Scope (Select (user.*, u, isFamily (u, Smith)); u2)
{
  Upon EnterHouse (u2, SmithHouse)
  Do Notify (Select (user.*, u3, isFamily (u3, Smith) AND u3 <> u2 AND isInHouse (u3,
SmithHouse)), “User ” + u2 + “just entered the house”)
}
```

1.9.2 Examples

We have defined an XML schema representing the ECA-DL syntax described in the previous sections (see Appendix A). This schema allows the validation of ECA-DL rules written as XML documents. In this section we present scenarios and we exemplify how to write rules using the XML schema. All the following examples are supported by the current implementation of ANS.

Example 1: Notify Maria when Jerry enters the kitchen.

In ECA-DL:

```
Upon EnterTrue (isInRoom (Jerry, Kitchen))
Do Notify (Maria, “Jerry has entered the kitchen)
```

In XML, this would look like:

```
<ECARule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ECAXML.xsd">
<upon>
	<event name="LocatedisInRoom" state_transition="EnterTrue">
		<param>
			<literal value="User.Jerry"/>
		</param>
		<param>
			<literal value="Kitchen"/>
		</param>
	</event>
</upon>
<do>
	<notification name="Notify">
		<param>
			<literal value="Maria"/>
		</param>
		<param>
			<literal value="Jerry has entered the Kitchen"/>
		</param>
	</notification>
</do>
<lifetime value="always"/>
</ECARule>
```

Example 2: Notify Jerry if he leaves his house and not take his laptop.

In ECA-DL:

```
Upon EnterTrue Located(isInRoom (Jerry, street))
When LocatedInjerrylaptopisOnline (computer)
Do Notify (Jerry, “ Jerry,  forgot your laptop”)
```

In XML, this would look like:

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

<upon>
	<event name="LocatedisInRoom" state_transition="EnterTrue">
		<param>
			<literal value="User.Jerry"/>
		</param>
		<param>
			<literal value="Street"/>
		</param>
	</event>
</upon>
<when>
	<function name="LocatedInisOnline">
		<param>
			<literal value="Device.JerryLaptop"/>
		</param>
		<param>
			<literal value=”Home”/>
		</param>	</function>
</when>
<do>
	<notification name="Notify">
		<param>
			<literal value="Jerry"/>
		</param>
		<param>
			<literal value="Jerry, you forgot your laptop"/>
		</param>
	</notification>
</do>
<lifetime value="always"/>

</ECARule>
```

Example 3: Notify the client application whenever someone enters the office number 10.

In ECA-DL:

```
EnterTLocatedUser.*Office10Upon TrueToFalse (isInHouse (Jerry, SmithHouse))
Do Notify (ClientAppUser.* entered the office 10, “”)
```

In XML, this would look like:

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

<upon>
	<event name="isLocatedIn" state_transition="EnterTrue">
		<param>
			<literal value="Jerry"/>
		</param>
		<param>
			<literal value="User.*Office10SmithHouse"/>
		</param>
	</event>
</upon>
<do>
	<notification name="Notify">
		<param>
			<literal value="ClientApp"/>
		</param>
```

```
            <param>
                  <literal value="User.# entered the office 10"/>
            </param>
      </notification>
</do>
<lifetime value="always"/>
</ECARule>
```

Example 4: Notify Jerry when Maria and Peter entered the kitchen. In ECA-DL:

```
Upon EnterTrue (isLocatedIn (Maria, kitchen)) and EnterTrue(isLocatedIn (Peter,
kitchen))
Do Notify (Jerry, “Maria and Peter are in the kitchen”)
```

In XML, this would look like:

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

<upon>
      <and>
            <operand_a>
                  <event name="isLocatedIn state_transition=”EnterTrue””>
                        <param>
                              <literal value="User.Maria"/>
                        </param>
                        <param>
                              <literal value="Kitchen"/>
                        </param>
                  </event>
            </operand_a>
            <operand_b>
                  <event name="isLocatedIn" state_transition="EnterTrue">
                        <param>
                              <literal value="User.Peter"/>
                        </param>
                        <param>
                              <literal value="Kitchen"/>
                        </param>
                  </event>
            </operand_b>
      </and>
</upon>
<do>
      <notification name="Notify">
            <param>
                  <literal value="Jerry"/>
            </param>
            <param>
                  <literal value="Maria and Peter are in the kitchen "/>
            </param>
      </notification>
</do>
<lifetime value="always"/>
</ECARule>
```

4 Tutorial

1.10 Target group of this tutorial

This tutorial addresses any application developers wanting to take advantage of ANS in their project. Despite this general claim we were thinking of Amigo project partners designing applications for work packages 5, 6 and 7 as well. Please don't hesitate to give us feedback to improve this tutorial at the best.

1.11 Contents and goal of this tutorial

This tutorial contains a documentation of the first version of the Awareness and Notification Service. It explains application developers how to build applications that use ANS and why it is an advantage to use this service.

The different parts (ANS, interface, client) are described and afterwards there is a step-by-step-explanation how to build an application that uses ANS.

1.12 How to use ANS

The Awareness and Notification Service provides basic functionality required to develop applications that allow people and other applications to stay aware of any significant change in context with minimal effort.

When building an application the first step is to use Amigo service discovery mechanism to discover ANS. The main interface of ANS is called IManageRule. This interface allows an application to install notification rules. A notification rule defines in which context a notification is sent to the application. For example, an application might define a notification rule that notifies Maria when her kids enter home. Once this rule is set, ANS constantly evaluates this rule. As soon as the rule turns to true ANS notifies the application.

The interface IManageRule allows the subscription, removal, updates and querying of rules. The rules should be expressed using the ECA rule specification language described in section 4.

Installing ANS for the tutorial
Follow the procedure in chapter 2; in addition a prepared OSCAR profile is available (where) that installs all bundles including the example client. Please note that there is a 'simplified version' of ANS available that does not use CMS. If you are using this version then you do not have to install the CMS components, because events can be triggered without needing external input.

Configuring Eclipse to edit the client
The Eclipse profile should retrieve amigocore, cms and ans from repository but the client from a local path.

1.12.1 The ANS interface

The Awareness and Notification Service provides an interface that can be used by external applications. Please refer to Section 1.7 to get a detailed explanation of this interface.

1.12.2 The ANS Client

The ANS Client is an application example that shows how an application can access ANS via the Oscar OSGI framework. You can get the source code from [Amigo gforge repository]/ius/awareness_notification/ansclient.

Applications that use the .net based framework or a private solution, can still use ANS as long as they use the interface as it has been described. The example code for this tutorial can be found in Appendix B.

1.12.2.1 Executing the client

After you have installed and started both, the ANS OSGI service and ANS client bundles as previously described, the GUI of the ANS client will appear. You see a very simple interface that only provides the basic functions: Connect to ANS via the big button in the right, invoke ANS' methods via the remaining six buttons, and a text area that logs your actions and the result of them. Just have a try and test some of the functions to see how they work.

1.12.3 Writing an own application

In the following the development of an exemplary ANS application on the basis of the Amigo ANS client is described. Application developers can extend the basic functionality of this client or replace several components with their own ones.

At first you have to understand the different tasks an application has to perform. It has to communicate with ANS in two ways:

- invoke its methods to subscribe rules, unsubscribe them, start them etc
- provide an interface enabling NS to notify your application if one of the rules you advised ANS becomes true

The communication runs via the webservice infrastructure of the Oscar framework, so you don't have to care about implementation details. Actually nearly every communication and infrastructural work is already done in the client project; you just have to paste your own application into it at a specific point. However, it is good for you to have a look in the code to understand how all this works. The amigo framework is explained in [Amigo_D3.4]

The main class is "MainComponent". With this class, the bundle is activated and the client service is exported. In this example, the service functionality is contained in the client.application.ClientNotify class. As you will see, this class provides a very basic way of informing a user. If you want to extend this, just add your own code to this class or replace it.

To manage communication in the other direction (invoke the IManageRule interface), you can use the class called "ANSWrapper". It encapsulates the ANS and provides every functionality of it to your application. To use it, just gain an instance of it by the ANSWrapper.getInstance() method, invoke connect() to search for ANS via service discovery automatically, and use the methods of this wrapper just in the same way as the methods of IManageRule. Please have a look in the GUI and the ANSWrapper's code to see it more detailed.

Here is a step-by-step-explanation how to transform your own application to an Oscar bundle that uses ANS:

- *Write your code*
    - Implement a mechanism to receive a notification by ANS. For this purpose, you have to implement the ans.api.INotifyApplication interface and it's notify() method or extend the given code. This method is invoked when ANS notifies your application.
    - Write a notification handler that does the following:

- Intensity I=0 then nothing
- Intensity 1<=I<=5 lightbulb on
- Intensity 5<=I<=9 messagebox
- Intensity 10 messagebox and sound

Your method could look like this:

```
public class Client implements INotifyApplication {
      public void notify(String message, String userID, int intensity)
{

            switch(intensity) {


            case 10:
                  showAlertNotification(userID, message);
                  break;
            case 9:
            case 8:
            case 7:
            case 6:
            case 5:
                  showMediumNotification(userID, message);
                  break;
            case 4:


            case 3:
            case 2:
            case 1:
                  showLowNotification(userID, message);
                  break;
            case 0:


            }
      }

//……..

}
```

- Implement a mechanism to invoke ANS' methods. Use an ANSWrapper instance to search for the service and communicate with it. Of course you can also extend the given code of the GUI already contained in the project.
- Write code that does the following
  - Find ANS

```
ANSWrapper ans = ANSWrapper.getInstance();

if(ans.connect(){
//connection successful
} else {
//failed to connect
}
```

  - Read rule from file
  - Subscribe to ANS

```
Rule rule = <read rule from File…>;
try {
      int id = ans.subscribe(rule, "ANS_Client");
} catch (RuleFormatException e) {
      textArea.append("Rule could not be subscribed\n");
}
```

  - Wait for key
    - If key=='u' then read new rule and update
    - If key=='q' the unsubscribe and quit

- *Paste it into the base project*
  - The ready-made components of the project are contained in the amigo.client package (MainComponent, ANSWrapper, Activator). You should paste your own application into the amigo.client.application package. In the example, the GUI is decoupled from the rest and moved to a separate package. You must not stick to this structure, just change it however you like. Replace the ClientNotify class with your own class that implements INotifyApplication.
  - Change MainComponent.java: In line 94, change the code if necessary to create an instance of your own INotifyApplication implementation.
  - Start your own application. In the example this is done by showing the GUI in line 121.

You should also change the package names so that no conflicts with other applications can occur. Don't forget to edit metadata.xml and manifest.mf in the res directory when you do this. Also change the predefined name in manifest.mf to give your bundle an own name.

- Write a rule

Assignment: Write a rule expressing the following:

"When Jerry leaves the house and his lunchbox is inside the house notify him"

The rule should have looked something like example 3 of the previous chapter.

- Generate the context event

Provide example context event and let user create the event (this is only for the version with real CMS attached) If we use the example above then we should at least provide userlocation and objectlocation and we would need 2 contextsources (persontracker via mobile phone and fridge for example) refer to appendix C on how to do this.

- Check if it works

5 Assessment

1.13 Overview

The Awareness and Notification Service is part of the WP4 intelligent user services and has been fully implemented and tested during the Amigo project. It consists of several components that have been combined into one Oscar bundle for deployment.

A tutorial was provided at the INRIA meeting in Rocquencourt, on April 28th 2007. Several responses from application developers have been supported. Whenever it was necessary, the components of the ANS service has been further developed to fit all requirements. The latest stable version of the ANS bundle has been made available at gForge.

1.14 Role of ANS within the Amigo Framework

Work package 3

ANS was build on top of the lower Amigo middleware. This means, other services and applications can discover ANS by using the discovery mechanism of the Amigo framework. ANS itself takes usage of these methods for discovery and communication with other Amigo services provided by the middleware.

Work package 4

ANS uses the Amigo Context Management Service (CMS) to access context data of the smart home as well as it uses the Amigo User Modeling and Profile Service (UMPS) to access the preferences of the users.

Work packages 5,6,and 7

ANS is a building block for context-aware Amigo applications. A list of applications using ANS is provided in the next section.

1.15 Usage of ANS in Applications

The main outcome of the work on ANS is its usage in several Amigo applications. These are illustrated in the following table categorized by the work package they belong to:

Table 2: ANS and its usage in Amigo

ANS	**WP5**	**WP6**	**WP7**
Application	• Crisis Respond Application • Food Management • Comfort Systems • Health Manager	• Privacy Enforcement, • Monitoring Manager	• Activity Sharing (Poker Game)
Service	N/A	N/A	N/A

A simplified instance of an ANS client was implemented for demonstration purpose only. It will not be assessed individually, but within the evaluation of the WP4 demonstration scenario.

1.16 Developers Feedback

Until now, only informal qualitative assessment has been realized basing on the feedback received by developers of Amigo applications using ANS. The conclusions of this feedback are the following:

- Regarding functionality, the set of attributes provided by the ECA-DL language has been repeatedly extended to suffer all known needs.
- Regarding speed and performance, the overall system showed no weaknesses running on a standard PC among other system services and applications running at the same time.
- Regarding reliability and availability, there are no cases known to us, where the last stable version of ANS has been inaccessible to Amigo applications caused by internal errors.

1.17 Benefits of Using ANS

This section summarizes the main benefits of ANS in a short:

- Enables to rapidly develop applications that allow users to stay aware of significant context changes in their environment
- Provides an easy-to-use and expressive language (ECA-DL) that allows applications developers to state what they are interested in (rule-based approach)
- Provides notifications with appropriate rendering of intensity, based on a user's preferences and current context

Appendix A: ECA-DL Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2006 sp2 U (http://www.altova.com) by Patricia Dockhorn
Costa (University of Twente) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">

<xs:element name="ECARule">
	<xs:complexType>
		<xs:sequence>
			<xs:element name="scope" type="scopeType" minOccurs="0"/>
			<xs:element name="upon" type="eventExpressionType"/>
			<xs:element name="when" type="expressionType" minOccurs="0"/>
			<xs:element name="do" type="notificationsType"/>
			<xs:element name="lifetime" type="xs:string"/>
		</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="notificationsType">
	<xs:sequence>
		<xs:element name="notification" type="notificationType"
maxOccurs="unbounded"/>
	</xs:sequence>
</xs:complexType>

<xs:complexType name="notificationType">
	<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="eventExpressionType">
	<xs:choice>
		<xs:element name="and" type="eventBinopType"/>
		<xs:element name="or" type="eventBinopType"/>
		<xs:element name="event" type="eventType"/>
	</xs:choice>
</xs:complexType>

<xs:complexType name="expressionType">
	<xs:choice>
		<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="eventBinopType">
      <xs:sequence>
            <xs:element name="operand_a" type="eventExpressionType"/>
            <xs:element name="operand_b" type="eventExpressionType"/>
      </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:complexType name="eventType">
      <xs:sequence>
            <xs:element name="param" type="expressionType" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="state_transition" type="xs:string"/>
```

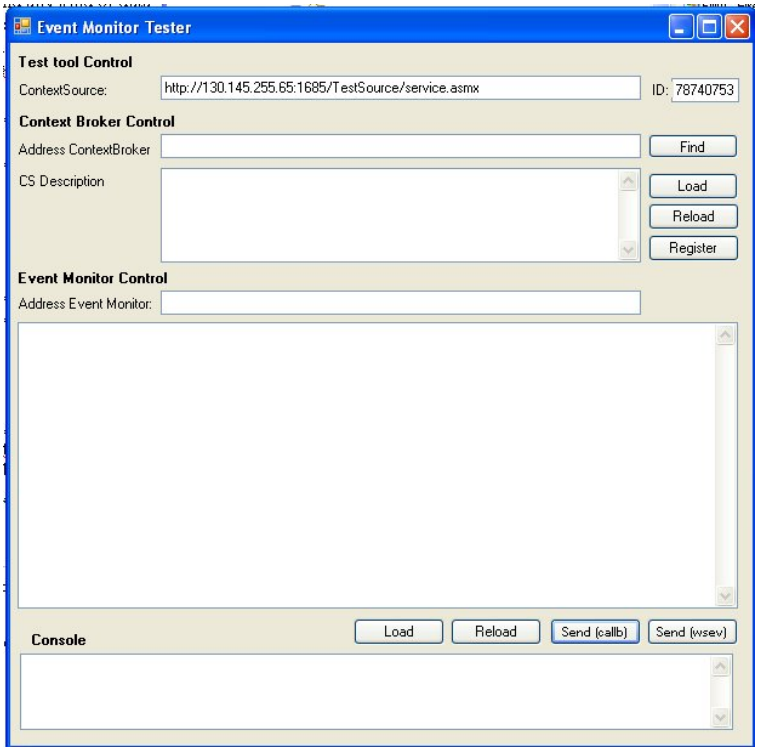
```
      <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

</xs:schema>
```

Appendix B: Code Example C#

```
using System;
using System.Collections;
using System.Text;
using System.Web.Services;
using System.Threading;

using EMIC.WSDiscovery;
using EMIC.WSDiscovery.Client;
using EMIC.WSDiscovery.Server;
using EMIC.WebServer;



namespace ANS_Client
{

    class ANSClient
    {
        private NotificationReceiver receiver;
         private XMLDocument rule;
         //this reference has to be created by adding ANS interface (wsdl file) to
       web references
         private ANS.RuleManager  manager;

        static void Main(string[] args)
        {
            Program client = new ANSClient();
            client.RunProgram();
        }

        public void RunProgram()
        {
             int subscriptionid = 0;
            bool ANSavailable = false;

            //Port = 0 gives an available port, start the receiver
            nr = new NotificationReceiver();
            WebServer.Port = 0;
            WebServer.AddWebService("ANSClient", nr);

             Myaddress = http://" + System.Net.Dns.GetHostName() + ":" +
WebServer.Port + "/ANSClient/service.asmx"
            Console.WriteLine("The ANS Client is now running, with url"+ myaddress );

             rule = new XMLDocument();
            rule.Load(“ANSTutorialRuleExample”);

             ANSavailable = findANS();
             if (ANSavailable)
             {
                    subscriptionid = manager.subscribe(rule, myaddress);
             }

            Console.WriteLine("Hit any key to quit");
            Console.ReadLine();
             If (ANSavailable)
             {
                    Manager.unsubscribe(subscriptionid);
             }
        }

       private bool findANS()
       {
```

```
            ProbeClient brokerProbeClient = new ProbeClient();
              ServiceScopeCollection brokerScope = new ServiceScopeCollection();
            ServiceTypeCollection brokerType = new ServiceTypeCollection();

            Console.WriteLine("Searching for ANS, this will take 5 seconds");
            brokerScope.Add(new ServiceScope("urn:amigo"));
             brokerType.Add(new ServiceType("ANS"));

                   brokerProbeClient.AddServiceScope(brokerScope);
                   brokerProbeClient.AddServiceType(brokerType);

                   brokerProbeClient.SendProbe();

                   Thread.Sleep(5000); //Wait 5 seconds

                   if (brokerProbeClient.ProbeMatches.Count > 0)
            {
                //Context Broker proxy, created from reference to wsdl
                manager = new ANS.RuleManager();

                //we do not check if this is a valid url
                //We will simply select the first instance of ANS we find
                manager.Url =
brokerProbeClient.ProbeMatches[0].FirstXaddr.ToString();
                Console.WriteLine("ANS found, address is: " + cbproxy.Url);
                    return true;
             }
             else
             {
                   return false;
             }

    }

    [ScopeAttribute("urn:amigo")]
    [TypeAttribute("ANSClient")]
    class Service : DiscoverableService
    {


        [WebMethod]
        public void receiveNotification(string message, int intensity)
        {
            Console.WriteLine("Received Notification"+message);
             //do something with it
        }
    }

}
```

Appendix C: EventMonitor Test Context Source



The event Monitor test context source is in fact a simple implementation of a context source that can send any kind of data.

Make sure that you have ANS and at least one context broker running[1], as explained in chapter 2. When you switch on the tool it should show its address in the corresponding text box (Adress ContextSource).

Step 1, search for the broker by pressing 'Find' Button

- when broker is found then its address is shown in box with the same name

Step 2, specify the context that you are 'providing'

- With old version of the broker this can be a string, examples of predefined context are "description_UserLocation.xml" and "description_UserActivity.xml" in the box labelled CS description
- With new version of the broker, this has to be in RDF format, examples can be found in userlocation.txt and useractivity.txt

Step 3, register by pressing register (you will see the result in broker console)

Step 4, call event monitor interface asking for known context (e.g. User, isLocatedIn Room or User, performsActivity, Activity)

[1] With current version of broker, there should be only one version, otherwise context might not be found

- The event monitor will subscribe to the test tool, this will become visible by a pop-up, after the pop-up, address of event monitor is filled in text box with same label

Step 5, load context information, examples can be found in sparqlanswer_userlocation.xml and sparqlanswer_useractivity.xml, via load button

Step 5b, change the context if you like (you can revert back to original via reload)

Step 6, send “context info event” to ANS via ‘send info’

- You should get an internal event

Appendix D: Example Rules

Rule example 1

```
<?xml version="1.0" encoding="UTF-8"?>
<ECARule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ECAXML.xsd">
	<upon>
		<event name="isLocatedIn" state_transition="EnterTrue">
			<param>
				<literal value="User.Luiz"/>
			</param>
			<param>
				<literal value="Close"/>
			</param>
		</event>
	</upon>
	<do>
		<notification name="Notify">
			<param>
				<literal value="ANS_Client"/>
			</param>
			<param>
				<literal value="Luiz is close to the demo site"/>
			</param>
		</notification>
	</do>
	<lifetime value="always"/>
</ECARule>
```

Rule example 2

```
<?xml version="1.0" encoding="UTF-8"?>
<ECARule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ECAXML.xsd">
	<upon>
		<event name="isLocatedIn" state_transition="EnterTrue">
			<param>
				<literal value="User.Peter_Vink "/>
			</param>
			<param>
				<literal value="Vicinity"/>
			</param>
		</event>
	</upon>
	<do>
		<notification name="Notify">
			<param>
				<literal value="ANS_Client"/>
			</param>
			<param>
				<literal value="Peter Vink is in the vicinity of the demo
site"/>
			</param>
		</notification>
	</do>
	<lifetime value="always"/>
</ECARule>
```

Rule example 3

```
<?xml version="1.0" encoding="UTF-8"?>
<ECARule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ECAXML.xsd">
	<upon>
		<event name="isLocatedIn" state_transition="EnterTrue">
			<param>
				<literal value="User.Richard"/>
			</param>
			<param>
				<literal value="Far"/>
			</param>
		</event>
	</upon>
	<do>
		<notification name="Notify">
			<param>
				<literal value="ANS_Client"/>
			</param>
			<param>
				<literal value="Richard Etter is far from the demo site"/>
			</param>
		</notification>
	</do>
	<lifetime value="once"/>
</ECARule>
```

References

Ami06a Amigo Deliverable D3.2: Prototype implementation. N. Georgantas (ed.), IST-004182 Amigo.

Ami06b Amigo Deliverable D4.2: Report on detailed Intelligent User Interface design, B Kladis (ed).

Ami06c Amigo Deliverable D3.3.