**Vrije Universiteit Brussel - Belgium**
**Faculty of Sciences**
**In Collaboration with Ecole des Mines de Nantes - France**
**and**
**Vrije Universiteit Brussel - Belgium**
**2007**



# User-adaptable context-aware applications in a mobile environment

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Mariana Hernandez Vazquez

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Stijn Mostinckx ( Vrije Universiteit Brussel )
Charlotte Herzeel ( Vrije Universiteit Brussel )

**Abstract**

Context-awareness is one of the key aspects of Weiser's vision on ubiquitous computing. In an ubiquitous environment, context-aware applications have the ability to sense and analyze context from various sources and use it to adapt their behavior. However most of the time, the decision logic, that determines the link between context and the adaptation it triggers, is hardcoded in the application base code. Users of context-aware applications can differ in terms of expectations about how their applications should behave and also they can move from one to another environment (hospital, house, street, etc.). Using a hardcoded approach, developers have to predict all these environments as well as all the end-users expectations. Even if developers could predict them all, environments evolve over time just as user expectations. Approaches like the GAIA and CRIME middleware enable the development of context-aware applications where the decision logic is seperated from the base code. However they still do not offer fine-grained support to involve the end-user in the development process. Another problem is that these applications act on behalf the user and as consequence he could feel like he is lossing control over their applications. In this dissertation, we argue for an accessible methodology to involve the user in the development of context-aware applications with an open ontology to describe the world. This methodology includes: developers' considerations that go beyond the applications itself and a description of the environment that enables evolution by means of composing and structuring data.

# Contents

# List of Figures

# Listings

# Acknowledgements

First of all, I would like to thank Prof. Dr. Theo D'Hondt for promoting this dissertation. I would like to thank to the people at the Ecole des Mines de Nantes for give me this opportunity.

I want to express all my thanks to Stijn Mostinckx and Charlotte Herzeel for proofreading my work and their valuable advice until the last moment. I also thank Jorge Vallejos for their comments, suggestions and support throughout my thesis research.

Diego, Alfredo, Rob, Toon and Pablo: I am grateful for their friendship. A special note of thanks to Annya Requile, Gabriela Arevalo and Pablo for their encouragements and emotional support.

Finally, I would like to thank my friends and family for being there.

Thank you.

# 1 Introduction

Weiser's vision of Ubiquitous Computing [Wei95] and the Ambient Intelligence vision [DBS+01] involves integrating sensors and microprocessors in everyday objects to make them smart. These visions predict a whole range of new possibilities, such as applications that alter their behavior depending on the dynamic context they are deployed in. However, novel, unforeseen problems arise during such applications' design. For example, these applications need means to explore the environment, the possibility to communicate with it and they should be able to assess in which way they can assist the users in their everyday tasks. In this dissertation we propose a methodology to involve the user in the development of context-aware applications.

## 1.1 Motivation

In Weiser's vision, context-aware application acquire context information from sensors and based on it, they decide which actions to perform. Unfortunately, most of the time the decision logic is hardcoded throughout the application's base code. As such developers anticipate all possible scenarios in advance to give them a fixed adaptation. This is acceptable for most adaptations triggered by nonhuman aspects, as these can be predicted by developers. Consider for example a mobile phone equipped with an accelerometer. This sensor makes it possible to detect when the mobile phone is rotated from portrait to landscape and viceversa. Thus, developers can associate the fixed action that determines their display configuration, based on the acquired sensor information. However developers are bad at encoding more human aspects and simply cannot predict all possible scenarios in the everyday life of each user. Therefore, the risk of doing the wrong thing, is quite high. To clarify the latter, consider for example Bob's mobile phone: *when Bob is with his boss, then all calls and messages should be redirected to his sec-*

*retary*. This small scenario shows that sometimes personal or social aspects need to be incorporated in context-aware applications. For this particular scenario for example, developers have to model formal relations between the application user and his boss. However the scenario only covers one such possible relation, we can come up with numerous others. On the other hand, the boss-employee relationship and the associated behavior of the mobile phone is also not of interest to many users. Clearly, not only nonhuman aspects dictate the user's desired behavior of the application.

Each user has different ideas on how the application should behave. This is because users interact in different environments or they perceive similar environments differently. In addition, environments and end-user expectations change over time, and context-aware applications should be able to evolve along with these changes. Hence following the commonly used approach of fixed scenarios with associated actions to model the application cannot possible cover all end-user expectations, nor does it support a dynamic evolution of the application. Examples of these kind of applications exist, such as a context-aware application for a museum that shows the content related to an exhibit when a user stands in front of it [AAH+97], or infers that near people wish to share their documents [DSA01], or in an office building, if someone moves between offices, he might wish to redirect all his phone calls to the nearest phone [WHFG92]. These examples infer human intentions and act implicitly without end-user considerations. For instance one user may not want to share all his documents since he considers them to personal. A user that wants to redirect all his phone calls except some and he may not want to disturb certain people in their office with his calls. We can think about different scenarios using the same action (redirecting calls or share documents). As a consequence end users have to adapt their behavior to what their context-aware applications can do for them and not the other way around. Also, given that this kind of applications act implicitly, end-users may feel less control over their applications as it was shown in [BD03].

The observations presented so far argue for a design of context-aware applications such that end-users themselves are allowed to configure them to better suit their personal needs. This will in general increase the acceptance of context-aware applications. However most end-users do not know how to program.

*In this dissertation, we argue for an accessible methodology to involve the user in the development of context-aware applications with an open ontology to describe the world.*

## 1.2 Dissertation Roadmap

The following chapter gives a general overview of Logic Programming as it constitutes the basis of CRIME, the middleware with which we perform our experiments.

Chapter 3 describes common elements in selected context-aware systems (Context Toolkit, JCAF, WildCat, GAIA). It also discusses and points out the current

mechanisms to reason about context information.

In chapter 4 we first describe the four principles (fluidity, intuitiveness, robustness and calmness) that underly organic user interfaces (OUIs). Afterwards, we present different approaches whose aim is to involve users in their context-aware applications. We analyze them based on the principles of OUIs. From these we take some important principles to include in our methodology.

Chapter 5 describes CRIME which is the middleware which we extended, we describe the Logic Coordination and the Fact Space Model that are the basis of this approach.

To illustrate our methodology, we present our CRIME extension to provide more flexible mechanisms to describe the world that can be understandable not only for developers but also for users in chapter 6. This CRIME extension adds composition and structuring to the current CRIME facts. During this chapter we describe how this extension enables to support environment's evolution in CRIME.

Chapter 7 describes the set of principles in our methodology that enables end-users to reason for themselves about their environment and decide which actions to associate. And describes the problems that arise with the rule definition: rules to general or to specific and/or conflicting rules.

The final chapter summarizes the results of this work and gives an overview of future work.

# 2 Logic Programming

In this dissertation, we argue for an accessible methodology to involve the user in the development of context-aware applications with an open ontology to describe the world. In this chapter we give a general description of the logic programming paradigm since it forms the base of CRIME, the middleware with which we performed our experiments. This description introduces terminology relevant for the rest of this dissertation. Also, we discuss the forward and backward chaining strategies. Further, since CRIME also implements the Rete algorithm in order to optimize the matching process, we present an overview of it in section 2.2.

## 2.1 Logic Programming

Logic programming fosters a declarative style, where programs indicate what to do, but not how. Logic programming is concerned with the examination, manipulation, and generation of knowledge. Such knowledge can be expressed with two basic constructs: facts and rules. In order to reason about these facts and rules a so called reasoning engine can use two reasoning strategies: forward or backward chaining. In this section we first describe the basic elements of a logic programming language. Subsequently, we discuss the differences between the two reasoning strategies.

### 2.1.1 Facts and Rules

Facts are unconditional statements that are assumed to be true or correct at the time they are introduced. In order to illustrate their use, we are going to use the London underground example from [Fla94]. In figure 2.1, we have different subway stations (e.g. Bond Street, Charing Cross, Green Park, etc.) and we see that they are grouped together over different lines (e.g. Jubilee, Central, Piccadilly, etc.). From the figure it is possible to identify different facts. For instance: *Bond Street station*

*is connected with Oxford Circus station on the Central Line*. During this chapter, we illustrate the concepts using Prolog which is the most well-known logic programming language [SS86]. This way, using Prolog syntax, this statement looks like `connected(bond_street,oxford_circus,central).` A fact has a name that denotes a relation (`connected`) and the arguments denote values from the problem domain (`bond_street, oxford_circus and central`) [Fla94].


Figure 2.1: London Underground [Fla94]

Listing 2.1: Fact examples

```
connected(bond_street,oxford_circus,central).
connected(bond_street,green_park,jubilee).
connected(green_park,charing_cross,jubilee).
connected(green_park,oxford_circus,victoria).
```

Facts are like data in a database. They are stored in a *knowledge base* and in order to reason about them, they must be asserted to it. *Asserting* data is to add the fact to the knowledge base, the term *retracting* data is used to signal that data is removed from the knowledge base. Looking up facts from the knowledge base is called *querying*, for example we can ask: *Are Bond Street station and Oxford Circus station connected?*, as a result we receive the facts that match the query, and false if there are not any facts that match.

On the other hand, rules are conditional statements, they describe facts that hold depending on certain conditions. Each rule has the form: *action* $\leftarrow$ *condition*. They can be seen as *if-then* patterns to be read as: if the *condition* is true or present, the *action* must be executed. The condition part of the rule is formed by patterns joined together by logical connectives. The engine applies rules to asserted facts, and we say that a rule is *triggered* if there are facts that match or satisfy the condition part of the rule. When a rule is triggered, the action takes place. Mostly, when we talk about an *action* in logic programming, we mean the assertion or the retraction of a fact from the knowledge base.

In the example so far, we have created a basic representation of the London underground. Now consider for instance that we want to specify all the nearby stations. It is defined that two stations are nearby if they are on the same line with at most one other station in between [Fla94]. For instance Bond Street station is nearby Oxford Circus station. Using Prolog syntax, this might look like: `nearby(bond_street,oxford_circus).`

Of course, it is possible to define and assert the nearby relation one by one. In practice, it may not be feasible to explicitly specify all the nearby stations, because they can be to numerous to list them all. A better way is to define rules that derive which stations are near to others, such as those given in listing 2.2.

Listing 2.2: Rules Examples

```
nearby(X,Y) :-
        connected(X,Y,L).
nearby(X,Y) :-
        connected(X,Z,L),connected(Z,Y,L).
```

We define two rules in order to describe how to find nearby stations. The first rule indicates that two stations are near each other when there is no intermediate station, while the second rule helps to derive when the stations are nearby with an intermediate station (Z) between them. In Prolog we define rules, first indicating the conclusion or action and after the conditions separated by the symbol ":-", it can be thought to mean "if" or "is implied by". The conditions can use comma "," to mean "and" operation. We can read the first rule as "it is true that there is a nearby station from X to Y if it is true that X and Y are connected in the line L". While the second says that to find a nearby station (X,Y) that are not directly linked, there must be an intermediate station Z from which is connected to Y and X, Y and Z are in the line L. In Prolog syntax, variables start with uppercase character, for instance in the example we have as variables X, Y, Z and L. After defining these rules, we can query about nearby stations, for instance: `nearby(bond_street, leicester_square).`

### 2.1.2 Backward Chaining

In the backward chaining strategy, the reasoning engine starts with the desired conclusion(s) or goal, and looks for rules that will help to find supporting facts. It is therefore called *goal-driven reasoning*. Prolog uses a backward chaining reasoning engine. The backward chaining tries to find supportive evidence (facts) for a goal. We can summarize the strategy, as follows:

In order to prove a goal G:

- If G is in the knowledge base of facts, it is proven.
- Otherwise, find a rule which can be used to conclude G, and try to prove each of that rule's conditions.

Consider now the that we have two simple rules defined as:

Listing 2.3: Backward chaining example

```
is_an_amphibian(Name) :-
        is_a_frog(Name).

is_a_frog(Name) :-
        is(Name,animal),
        is(Name,green),
        can(Name,jump).
```

Using a backward chaining strategy, we start from a set of goals. Consider that our knowledge base contains the facts presented in figure 2.2. Then the cycle starts from the goal that *Kermit is an amphibian*. Since there is no fact that matches this goal, the reasoning engine searches for a rule which derives this as a conclusion. Then it adds the condition of the rule as a new goal to be proven. Thus now the new goal is *Kermit is a frog*. Following the same principle, since there is no fact in the knowledge base that says that Kermit is a frog, it will search again for a rule, then we will get more goals to be proven, namely that *Kermit is an animal, it is green and it can jump*. Finally each one of these goals can be proven one by one based on the available facts.

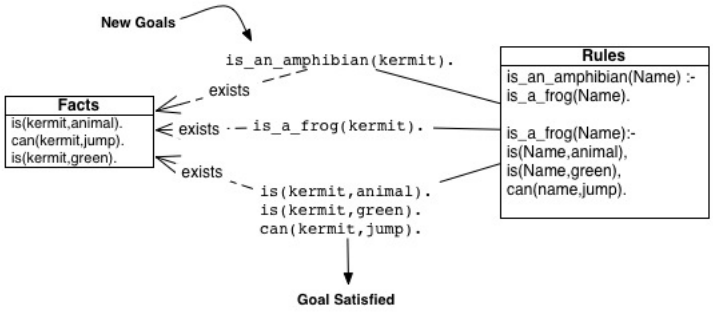


Figure 2.2: Backward chaining

### 2.1.3 Forward Chaining

Forward chaining starts with the facts and applies rules to find all possibles conclusions. Since the facts (data) available in the knowledge base determine which rules are used, this method is also called *data-driven*. We can summarize this algorithm, like:

- Select a rule whose conditions match the facts stored.
- If there is more that one rule selected, we call them the conflict set, then the engine uses a conflict resolution strategy (e.g. priorities) to select one.

- Execute the action or conclusion of that rule, thus somehow changing the current state.
- Repeat until there are no rules which apply.

We apply this strategy to the frog example discussed before. The reasoning engine starts looking for a rule that should be triggered based on the current facts, as shown in figure 2.3. The rule engine contains the facts that indicate that Kermit is a green animal and that he can jump. Since this conditions triggers the rule that indicates that Kermit is a frog, and this fact is asserted to the knowledge base. Again the reasoning engine looks for a rule that should trigger with the facts in the knowledge base, notices that it contains a new fact which triggers that Kermit is an amphibian. This fact is therefore asserted to the knowledge base and since there are no more rules triggered with the current facts the reasoning engine stops.

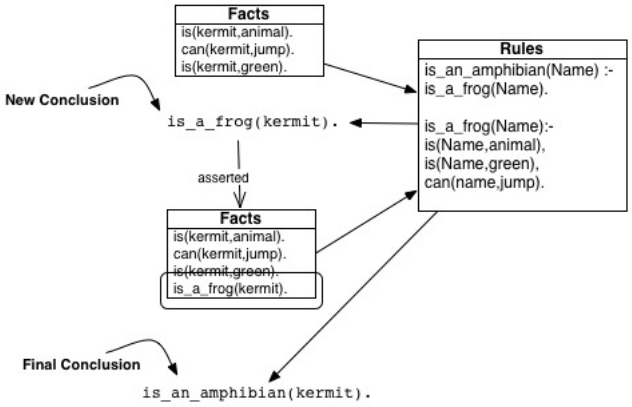


Figure 2.3: Forward chaining

### 2.1.4 Pattern Matching and Unification

In the previous sections we informally introduced terms like matching, in order to determine which rules trigger. Here we give more detail about this process.

**Pattern Matching** is the process where we compare a fact with a specified pattern in a rule condition in order to see if they match. In order to do that, a pattern matcher receives the fact, the pattern and a frame. A frame is a set of variable bindings, it enables to receive and return the mappings of variables to values. For instance, consider that a pattern matcher receives `connected(bond_street, oxford_circus, central)` as data, and we want to match with `connected(X, Y, L)`. The result should be a frame with the corresponding bindings, as shown in figure 2.4.

Now consider the pattern used in the example would have been `connected(X, Y, victoria)`, then the result of the pattern matcher is that the match has failed.

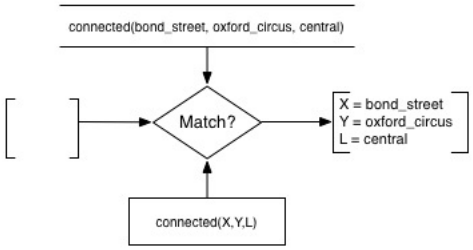


Figure 2.4: Pattern Matching

From the example we can see that variables match with symbols and then they are added to the frame, and symbols must be equal otherwise the match fails.

**Unification** is a generalized case of pattern matching in which both pattern and the data may contain variables [AS96]. Consider two patterns and an empty frame, figure 2.5. The process is the same as the pattern matching. The main difference is that we can match variables to variables, whereupon we can have variable bindings with variables or variables with undetermined values. In subsequent steps the bindings for these variables will need to be identical.

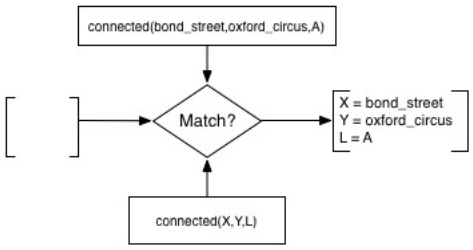


Figure 2.5: Unification

## 2.2 Rete Algorithm

In this dissertation we build upon the CRIME middleware presented in chapter 5, which employs a forward chaining inference engine based on the Rete algorithm which is outlined in this section.

The Rete algorithm was introduced by Charles L. Forgy [For82]. The algorithm was implemented to be used in a production system, using a forward chaining. In Production systems rules trigger actions which typically go beyond asserting and retracting facts. A production system has three basic elements: a working memory that contains the current state in the form of facts, the knowledge base which contains rules, and an interpreter that is in charge of the repetitive process of

matching, resolving the conflict set and performing the correct actions. The Rete network reacts when there are changes to the working memory or in the knowledge base. Each time an element is added to the working memory, the Rete network is updated in order to check which new conditions are met.

The aim of the Rete algorithm is to optimize the pattern matching process which is done by creating a network representation of the rules and by caching intermediate results. The purpose of the caching is to avoid the iteration over the working memory every time a fact is asserted or retracted from the working memory. This way, there is no need to recompute previous results.

**Compilation process** To explain the compilation process we review the example presented in section 2.1.2, where we have the following rule:

```
is_a_frog(Name) :-
        is(Name,animal),
        is(Name,green),
        can(Name,jump).
```

The compilation process of the network is performed by two complementary parts: the *alpha network* and the *beta network*. Each network has data filters and these are represented in the network as nodes. The Rete network works with tokens. A *token* is a tag with a list of working memory elements. The tag represents assertion or retraction from the working memory, indicated by + and - respectively. Tokens are transmitted to the node's children if they pass the filter of the corresponding node. Examples of tokens are in listing 2.4.

Listing 2.4: Tokens example

```
< + is(Kermit, animal) >
< + is(Kermit, green), is(Kermit, green) >
```

Tokens are saved in memories attributed to the nodes in the network to enable the caching of intermediate results. The complete network representation of the previous rule is presented in figure 2.6.

### 2.2.1 Alpha Network

The alpha network consists of filter nodes. New nodes are created to filter different kinds of data which are part of a prerequisite in the rule. Thus the alpha network has filter nodes to filter the type of the working memory element. In our example we have a filter node for the "is" and "can" type. Also new filter nodes are created to filter symbol elements stating that for instance the argument in the first position must be equal to "jump". In figure 2.6 we present the alpha network representation of our example. Observe that each filter node in the alpha network has a memory in order to do the caching. These memories save tokens that pass the filter of the associated node.

The alpha network also includes filter nodes responsible to filter variables occurring more than once within a single element. Consider that we have a prerequisite: `prerequisite(X, X)`. Here it must check consistency between the values of the arguments. For instance the filter node, for `prerequisite(X, X)`, matches the fact `prerequisite(abc, abc)`, but not `prerequisite(abc, qwe)`.

### 2.2.2 Beta Network

After the creation of the alpha network, the rule compiler creates nodes that combine two different branches. The branches are combined by means of *join nodes*. These join nodes are responsible to check consistency between variables in different prerequisites of a rule. In join nodes we work with more than one prerequisite. These kind of nodes have two memories: right and left memory, one for each branch that a join node combines. Each memory is activated when it receives a token from the previous node. In figure 2.6 we show the beta network for our example. Thus since the first and the second prerequisite of the `is_a_frog` rule share the variable for the argument at position zero. For instance, if we insert a token `<+ is(Crazy, animal)>` the left memory of the join node is activated and it is combined with the right memory to check the consistency between the value (`Crazy`) of the variables (`left.name = right.name`). However, since initially the right memory is empty, there is no checking and no token to pass. When subsequently a second token is inserted with `<+ is(Kermit, green)>`, the right memory of the join node is activated and the join node checks that Crazy and Kermit are different, since the variable is not consistent there is no token passed. However when a third token is inserted `<+ is(Kermit, animal)>`, the join node combines the left and the right memory passes the new token (`<+ is(Kermit, animal), is(Kermit, green)>`) to its children.

After the last join node of the beta network, it is added a terminal node, which corresponds to the production. When a token arrives to it, the action or the consequences of the rule should be executed.

The root node is a simple node that is in charge to distribute the tokens over all the different filter nodes that form the alpha network.

**Processing a fact** When a new fact is asserted, a new token is created and inserted in the root node of the Rete network. Then the root node distributes the token to all its children, in the alpha network. Each node it is responsible to filter the token: only if the token satisfied the node condition, it will be saved in the memory and passed to the children nodes. When a token arrives to a join node, it will be tested against the opposite memory from which it arrives, forming new tokens. When a token arrives to the production node, it means that it has passed all the conditions, and the action should be executed.

Setting back to the example, the Rete network presented in figure 2.6 also exemplifies the state of the various memories when we have the following in the

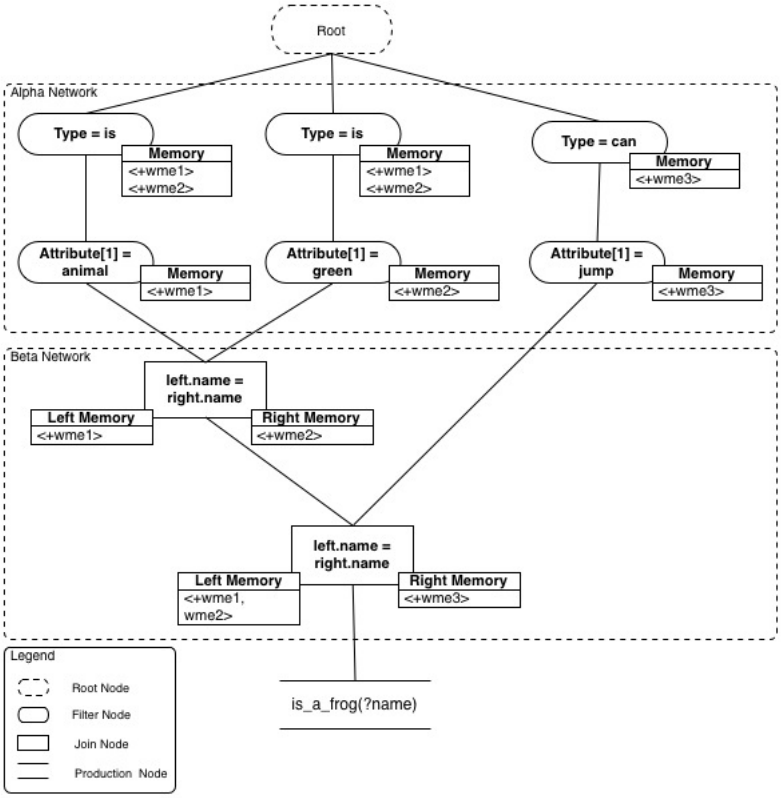


Figure 2.6: Basic Rete algorithm - Network Representation

working memory:

```
Wm1 is(Kermit, animal)
Wm2 is(Kermit, green)
Wm3 can(Kermit, jump)
```

Furthermore, the Rete algorithm enables to shares nodes, these nodes can be shared by the same production or by different productions. In the previous example, there are two identical nodes (`Type = is`) which can be simplified like in the figure 2.7 where we show the alpha network of our example. The sharing of nodes reduces the memory usage of the algorithm.

## 2.3 Conclusions

In this chapter we discussed logic programming which underlies the CRIME middleware in which we conduct our experiments, which is described in the following chapters. We have presented the basic terminology in logic programming. We also explained the basic Rete algorithm which is implemented in CRIME. The

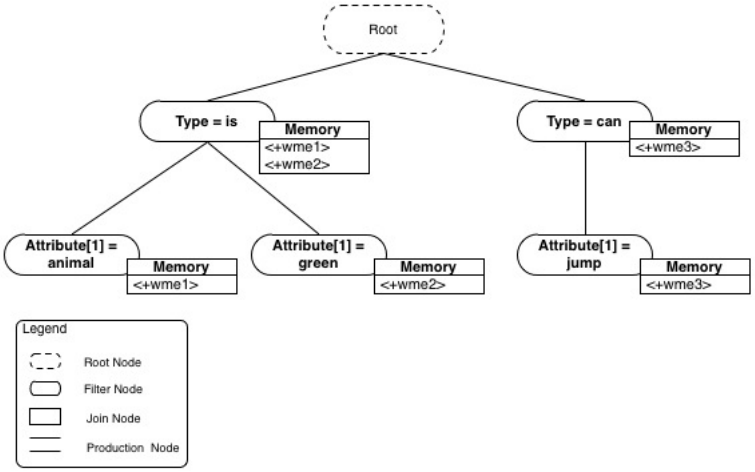


Figure 2.7: Shared nodes in Rete Network

Rete algorithm is an optimization of the forward chaining pattern matching. In the following chapter we review representative context-aware systems for creating context-aware applications. In addition we discuss if and how these systems allow application users to configure their context-aware applications.

# 3 Context-Aware Systems

In this dissertation, we argue for an accessible methodology to involve the user in the development of context-aware applications with an open ontology to describe the world. The previous chapter gave an overview of the logic programming principles, underlying the methodology we propose in the remainder of this dissertation. In this chapter an overview is provided of some classical context aware systems. Context aware systems enable the development of context-aware applications. This kind of applications aims to sense context information and modify the behavior according to predefined expectations. We analyze context aware systems based on their support for three common elements: context acquisition, context sharing and context reaction.

This chapter first gives an introduction to context and establishes the common elements of a context-aware system. Subsequently, some context-aware systems are presented: Context Toolkit, JCAF, WildCAT, Gaia and LIME. To conclude, an evaluation of these context-aware systems is presented.

## 3.1 Context

The first research on context-aware computing was the Olivetti Active Badge, whose goal was to redirect calls to the nearest telephone, based on user location [WHFG92]. The term context-aware computing was introduced by Schilit and Theimer in [SAW94]. Since then different definitions of context have been given. We can refer to the Cambridge Advanced Learner's Dictionary[1] where context is defined as:

> The situation within which something exists or happens, and that can help explain it.

[1] http://dictionary.cambridge.org/

However this kind of definition is to unprecise to be useful in the field of computer science. Thus, other explicit definitions of context are often given in terms of an enumeration of different kinds of contexts such as: location, nearby people and devices, light level, network connectivity, time, even social situations [SAW94]. However, such an exhaustive definition is not sufficient: when we are facing a new situation where the information is not included in the list, it becomes vague if we can see it as context or not. Another point to be taken into account is that sometimes some information like for example the physical environment of an application, is considered as context, whereas in other situations it is deemed irrelevant. In the latter cases it might be confusing if such information is listed.

Therefore we consider a more appropriate definition the one given by Dey in [ADB+99]. It is an operational definition where context is defined as:

> 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.

This definition is more general, and gives some freedom to the developer. Developers are able to determine which information they consider as context.

In order to develop context-aware applications, different context-aware systems have emerged. These context aware systems share some properties. We can identify at least three common elements: context acquisition, context sharing and context reaction. Context aware systems typically specialize in one or more of these characteristics. To better understand each characteristic, we illustrate each by discussing the following scenario:

*Every morning Bob drives to his office. Since driving requires absolute attention on the road, his Pda (Personal Digital Assistant) re-routes all his incoming calls to voicemail. Also, his Pda redirects the sound of the music player to the car's audio system. The car's audio system starts playing his favorite music. Bob also has predefined volume preferences: when he drives alone, he likes to hear his music loudly. However sometimes he gives a ride to his colleague Alice. Since he prefers to talk with her during the way, he has a preference stating that when he is with a colleague, the volume should be lowered.*

### 3.1.1 Context Acquisition and Representation

According to [Che03] a context-aware system can acquire context information directly from sensors, by means of widgets or using a centralized server of context. We can have hardware and software sensors. Whereas hardware sensors are responsible to acquire physical context information, software sensors get context information from other applications like agendas, emails and even user inputs. Acquiring contextual information directly from sensors, especially in the case of hardware sensors, can be cumbersome since very often such sensors produce low-level

information. For instance, a location sensor, reports coordinates, sensor id, time of the sample, etc. In order to separate the application code from the technical details to deal with the management of the hardware, most of the context-aware systems provide a mechanism to factor out the technical details of context acquisition from the application. This mechanism is called *widgets*. Another approach involves the use of a centralized server,which gets all information from the different sensors and when some application needs contextual information, it has to query this server.

Interpreting a piece of low-level information to obtain high level information is an other way to acquire context information. For instance, in the scenario presented above, with the use of an interpreter we can get a person's name (Bob or Alice) from an RFID sensor that retrieves the user's id.

Furthermore it is possible to aggregate or compose different pieces of low-level information. In our running example, in order to detect that Bob is driving his car, we require context information from multiple sensors. One sensor detects the weight on the driver's seat, another detects whether the engine is running and a last sensor is used to determine Bob's location.

### 3.1.2 Context Sharing

Given that context information is distributed over different mobile and fixed devices connected via a computer network, it is essential to share this information to give each device a consistent view of the environment. Different approaches have been proposed. One approach involves using a centralized server that maintains context information for all devices that are able to communicate with it. However, devices in a context-aware environment, a centralized approach is not convenient. Using peer-to-peer communication, two devices can communicate directly without the need of an intermediary. Finally using a *shared memory*, such as a blackboard, devices can be notified about context changes that they are interest in.

For example, in our running example, Bob's pda needs to know that Bob is driving. Since the driver's seat's sensor is part of the car and not of Bob's pda, a mechanism for sharing information between both devices (the car and the pda) is required.

### 3.1.3 Context Reaction

Context Reaction is related with the adaptation of the system. It is an important characteristic since it implies the ability to react to changes in the environment. This reaction or adaptation takes place when some conditions in the environment are met. Often the process of context reasoning takes place in the application itself, as a *context client*. A context client is an application that adapts or reacts based on changes in its context. Since the reaction logic is commonly situated inside the code of the application, this situation leads to applications with diverse `if-then-else` statements manually encoding the different scenarios. Other approaches have emerged where conditions and reactions are encoded as rules, i.e

using a declarative style by means of logic programming.

Reviewing our example again, we can identify different conditions and associated reactions. First, in order for Bob's calls to be rerouted to the voice mail, the condition that he should be driving must be met. The same condition has to be met in order to redirect the sound of the music player. Volume adjustments are based on the presence of a colleague.

So far we have given a high-level introduction to context-aware systems and named the commonalities: context acquisition, context sharing and context reasoning. We next present an overview of how the latter are integrated in existing frameworks and middleware for implementing context-aware applications. First of all, we explain how the context toolkit enables context acquisition and how it represents context information. Secondly, we describe how the context is shared and thirdly we discuss the context reaction provisions. Finally we make some concluding remarks. We apply this kind of analysis to each approach presented.

## 3.2 Context Toolkit

The Context Toolkit described in [SDA99, DSA01], focusses on providing a mechanism called *context widgets* for context acquisition. These context widgets are inspired by GUI widgets which are pieces of reusable software that hide and manage details of physical interaction between a user and an application. A GUI widget allows using different devices (mouse or keyboard) to get user input without requiring any changes in the application. The result of this interaction is a callback from the widget to the application. GUI widgets act like a layer of abstraction between the application and the user, and in the same way context widgets abstract the interaction between the application and the environment.

### 3.2.1 Context Acquisition

There are three components of the Context Toolkit responsible for context acquisition: widgets, aggregators and interpreters. Widgets are the primary step to get low-level context information, whereas aggregators and interpreters constitute a source of new high-level context information based on widget information.

#### Widgets

A context widget is a reusable software component that hides physical details of the environment from the application. This is done by wrapping sensors with an uniform interface. Reusability comes from the fact that a context widget can be used by different *context clients*.

The Context Toolkit also introduces a special kind of server. A server provides access to different widgets of the same kind, since they provide related information. It also includes a privacy manager which acts as a security access filter, ensuring

that only context clients with the connect permissions have access to this context information.

Context information is represented as the state of a context widget which is a set of attributes modelled as key-value pairs. These key-value pairs are accessible by applications. An application has two ways to access the contextual information. One way is to register with the widget to be notified when context changes are detected. This way, it is also possible to specify some conditions that have to be satisfied in order to only receive filtered information and keep irrelevant information out of the application. The second form is by directly querying the widget, asking it for the context information.

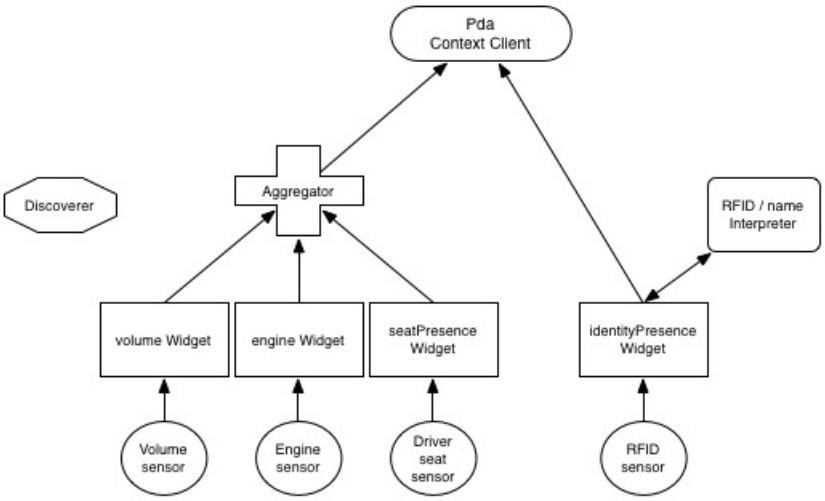


Figure 3.1: Context Toolkit - Context Architecture

In our example we can identify the different widgets as shown in figure 3.1. The figure contains one widget for each sensor that we have identified in our scenario.

### Aggregators

Aggregators aim to collect related context information. In this form an application does not need to have access to each widget that contains relevant context information, instead it just has to communicate with the corresponding aggregator.

We use an aggregator as part of our example in order to determine that the user is driving. The driving aggregator receives context information from the `engine` widget, the `seatPresence` widget and the `identityPresence` widget, in order to determine if the user is driving.

#### Interpreters

A context on interpreter abstracts low-level context information into high-level context. For instance, an interpreter can receive a user ID, and based on this information retrieve available user information like his name, telephone, address, etc.

As we can see in figure 3.1, location information is gathered by the RFID sensor. The output of the RFID sensor is passed to the IdentityPrecense widget, which is able to determine the person's name by using the RFID to name interpreter.

### 3.2.2 Context Sharing

In order to provide a common communication medium between all the components in the Context Toolkit the Extensible Markup Language (XML) is used over HTTP protocol in a peer-to-peer architecture.

Since sensors can be located on different devices on different locations, widgets are already physically distributed over the network. Applications are able to share context information by having access to the same widget by being notified when context changes or by querying the widget to ask for context information. However, in order to establish communication with a specific widget, interpreter or aggregator, the application uses a *discovery component*.

#### Discovery server

The discovery process relies on a centralized way to contact a specific component. As a consequence every component must be registered with the discovery server in order to be able to contact it. Also, each component must indicate what information it provides an how it can be contacted. Applications as context clients can query the discovery server by the name of the component which they want to communicate with or they can ask for specific attributes of the component. This way applications get a handle to communicate with the required component.

### 3.2.3 Context Reaction

Context Toolkit provides events in order to be aware that the context has changed. However reasoning about these changes is left to the context clients that make use of the context service in order to make an adaptation.

#### Context Service and Actuator

Context clients determine which actions should be executed by means of context services is in charge to control or change the current environment through the actuators. Context services and actuators are reusable by other applications that deal with the same kind of adaptations. Actuators are responsible to make a change in the environment.

### 3.2.4 Conclusions

The Context Toolkit covers all the characteristics presented in section 3.1. However it is mostly concerned with the context acquisition, by means of context widgets. Context reasoning resides in context clients which most of the time leads to if-then statements to encode the different scenarios.

For consistency we are going to use the Context Toolkit terminology as a reference in the subsequent explanation of context-aware systems.

## 3.3 JCAF

The Java Context Awareness Framework (JCAF) [E.05] is in the same field of the Context Toolkit, since it is a Java-based framework for development of context-aware applications. Like the Context Toolkit, JCAF is event-based. One of the main differences between Context Toolkit and JCAF is the explicit representation of entities such as places, persons, etc. and the relation with their context in an encapsulated component. JCAF uses a single component to access and change contextual information. As much context clients not only are able to be notified when there are changes in the context, they are also able to change it using a single component.

### 3.3.1 Context Acquisition

We can identify that there are some differences between the Context Toolkit and JCAF the most important being that JCAF introduces an explicit representation of entities.

#### Entity

In the definition of context in section 3.1 we already mentioned that entities can be places, persons, etc. JCAF always relates context to entities which are grouped in an Entity Environment.

Context information is represented as in the Context Toolkit by attributes with key-value pairs. However, JCAF provides a single component to access context information of a specific environment.

#### Entity Environment

As we already mentioned, JCAF groups entities and their context in a specific environment. In our example presented in section 3.1 we identified that the automobile itself is an specific environment, which could be represented as a JCAF entity environment. The entity environment does not only contain entities: it also provides *access control*. *Access control* is used to authenticate each *context client*, to provide a degree of security. The entity environment also includes the related aggregators and interpreters.

#### Widgets

Widgets in JCAF are used as in the Context Toolkit to get context information from sensors. Widgets capture changes in an entity's environment. Since context clients in JCAF are able not only to be notified about context changes but also to change entity's context information, widgets are considered as an special kind of context clients.

As in the Context Toolkit, context clients obtain context information by asking for it, or are automatically notified of changes by subscribing as an entity listener. However JCAF allows type-based subscription, where a client will be notified when a change occurs in an specific type of entities.

#### Aggregators and Interpreters

Aggregators and interpreters are implemented by means of an interface, shown in listing 3.1, where the only difference between an interpreter and an aggregator is the number of elements in the array of the translate method. Interpreters accept a single element, whereas an aggregator accepts two or more elements.

Listing 3.1: JCAF - Interface for interpreters and aggregators

```
public interface ContextTransformer {
        public Class[] getInType();
        public Class getOutType();
        public ContextItem translate(ContextItem[] in)
                throws ContextTransformerException;
}
```

### 3.3.2 Context Sharing and Reaction

Context sharing in JCAF is organized in a peer-to-peer setup, where different entity environments can query each other in order to share context information. In order to get context information from other entity environments, the client must know the id of the other entity environment, since this id constitutes a parameter of the `lookupEntity` method.

Entities are responsible to respond to changes in their context, when they are subscribed to some specific entity and context. There are no important differences between Context Toolkit and JCAF: both provide a Context Service that works together with an actuator to change or to adapt the current environment. However in JCAF context services are considered as a special kind of context client since they change the context of an entity in the entity environment.

In listing 3.2 we present the method `contextChanged` which is called each time that the context of the entity PDA changes. This means that whenever Bob's location or current activity changes, this method is invoked.

Listing 3.2: JCAF - Entity example

```
public class Pda extends GenericEntity {
```

```
 ...
 public void contextChanged (ContextEvent event) {
  if (event.getRelationship() instanceof Activity) {
   if (event.getItem() instanceof Action) {
     this.setActivity = ((Action) event.getItem()).getAction();
   }
  }
  if (event.getRelationship() instanceof Located) {
   if (event.getItem() instanceof Location) {
     this.setLocation = ((Location) event.getItem()).getLocation();
   }
  }
  if(this.activity == DRIVING && this.location ==AUTOMOBILE){
   Profile.redirectCalls(VOICEMAIL);
   Mplayer.out(AUTOMOBILE);
   Mplayer.start(FAVORITE);
  }
 }
 ...
}
```

This method keeps track of Bob's current activity and location. When the conditions that he is driving and that he is in the car, are met the PDA profile reroutes calls to the voicemail, the music player is requested to redirect the sound to the vehicle's music system and it starts playing his favorite music.

### 3.3.3 Conclusions

As in the Context Toolkit we see that JCAF provides a mechanism to access context information. The task of the programmer is to predict possible context usages and then make the adaptation that he considers to be adequate. However, dealing with adaptation in the application is part of the application's based code and not accessible or modifiable by end-users.

## 3.4 WildCAT

WildCAT aims to ease the creation of context-aware applications [DL05]. Similar to the previous approaches, it is event-based, which enables to represent context changes as an event. It is focussed on the representation of context as paths that enable to know both the values of a particular context parameter as well as its structure. WildCAT's paths introduce a hierarchical view on context information.

### 3.4.1 Context Acquisition

More than context acquisition, WildCAT focusses on the context representation by means of hierarchical paths.

### Context

Context is categorized in different domains. Each domain is a hierarchical structure of named resources where each resource has its context information modelled as key-value pairs.

WildCAT has four predefined domains: `sys`, `net`, `geo` and `user`. The `sys` domain refers to the hardware resources like input and output devices, `net` domain is concerned with network aspects, like bandwith and performance, `geo` is about geophysical information like temperature, or sound in our example, and `user` refers to the user aspects like preference of volume in our example, his activity, etc.

Paths represent resources, attributes of them or all sub-resources and subattributes [DL05]. The path syntax is :

```
domain://path/to/resource#attribute
```

Revisiting the example of Bob driving his car presented in section 3.1, in listing 3.3 we show some example paths to relevant context information. First a path for a location resource is listed. The second path represents an attribute of the volume resource. It is also possible to make a more general path: listing 3.3 illustrates how through use of "*" we can refer to all resources in an specific path or all the attributes for a particular resource.

Listing 3.3: WildCAT path examples

```
geo://location/input/rfid
geo://sound/volume#intensity
geo://sound/*
geo://sound/volume#*
```

Events are related with this path representation, implying that listeners can be registered to trap events such as adding or removing a new resource, changing an attribute.

Listing 3.4: WildCAT - Examples of EventListener

```
context.register(myListener, RESOURCE_ADDED | RESOURCE_REMOVED,
        new Path("sys://devices/output/audio/*"));
        ...
context.register(Expressionlistener CONDITION_OCCURRED
        "sys://devices/engine#state=on");
```

In listing 3.4 we have registered two event listeners to be notified of context changes. In the first example context clients are notified when a new audio output device is discovered, and the second one is related when a certain condition occurs, namely that the engine is working. This kind of listener allows filtering context information since context clients do not have to take into account *each* context change.

The creation of the initial structure and the changes to it is done by a XML configuration file. Listing 3.5 presents an example where we register a `driverSeat-Sensor` to the `sys` domain.

Listing 3.5: WildCAT - XML configuration

```
<context-domain name="sys">
    ...
    <resource name="driverSeatSensor">
        <sensor name="dSeatSensor" class="SeatSensor">
            <schedule><periodic period="5000"/></schedule>
        </sensor>
    </resource>
    ...
```

#### Widgets

Widgets are responsible for gathering the context information. WildCAT provides a centralized manager component where all sensors have to be registered and where they send their context information. Thus context clients only have to communicate with the manager. WildCAT has two different kinds of widgets: active and passive widgets. The difference between these two is the required schedule policy. Active widgets send context information to the manager when it is necessary, whereas passive widgets have an interval of sampling, implying that they are periodically queried by the manager. In listing 3.5 we present an example of a passive widget where we can identify the sampling period for the widget.

### 3.4.2 Context Sharing and Reaction

Context clients can access context information in two ways namely synchronously or asynchronously. In a synchronous model, clients request and discover context by explicitly asking for the information. For instance in listing 3.6, we ask for all output devices. In an asynchronous model context clients must be registered as context listeners in order to get notifications about context changes. Listing 3.4 already exemplified this case.

Listing 3.6: WildCAT - Synchronous model

```
context.getChildren(new Path(sys://devices/output/*));
```

An important limitation of WildCAT is that it does not provide a mechanism to communicate between different devices. It must be done in an ad-hoc manner using XML configuration files.

### 3.4.3 Conclusions

Similar to the approaches presented previously, WildCAT does not address the problem that adaptations based on the context information remain tangled with the base application code. As a consequence all possible scenarios have to be preconceived during the development of such systems. Thus if our system does not have a predefined way to identify a colleague, the user is not able to add it to

the system. However, it provides a hierarchical mechanism to describe the context information, and the possibility to create relations between entities.

## 3.5 GAIA

Gaia focuses on a context model based on first order logic, where context is represented as facts [RC03]. With this approach it is possible to reason about the context by means of rules. GAIA enables distributed reasoning, where each device decides how to reason about its context.

### 3.5.1 Context Acquisition and Context Model

Context acquisition is tightly couple with the context model since this determines the form in which widgets must represent the context they detect. Here we present the context model used in GAIA.

#### Context Model

The context model of GAIA focuses on the use of first order logic. As we have presented in the previous chapter, a fact's label describes the kind of information involved. In GAIA it is used to distinguish the type of context. We present some examples of GAIA facts in listing 3.7.

Listing 3.7: GAIA - Fact examples

```
Location(Bob, in, vehicle)
Application(MP3 Player, is, starting)
Sound(output, to, vehicle)
Sound(volume, ">", 80 db)
```

GAIA uses an ontology with the purpose of providing a relational view between context types and required arguments. This approach restricts the number of fact arguments and consequently enforces a specific structure. Many facts based on the ontology follow the format: `Context Type(<Subject>, <Verb>, <Object>)` as is noticeable from the first three examples presented in listing 3.7. Also GAIA allows relational operators inside facts, like the last example in listing 3.7.

GAIA allows describing contexts by connecting different facts using boolean operations like AND, OR and NOT. As an illustration, we present one example of each in listing 3.8. The first one requires that both Bob and Alice are inside the vehicle. The second one requires that we received an e-mail or a call. And the last one that Alice is not in the vehicle.

Listing 3.8: GAIA - Booleand operations

```
Location(Bob, in, vehicle) AND Location(Alice, in, vehicle)
Receive(Mail) OR Receive(Call)
NOT Location(Alice, in, vehicle)
```

In addition, GAIA includes mechanisms to specify context using existential and universal quantifiers, as is shown in listing 3.9.

Listing 3.9: GAIA - Existence and Universal Quantifiers

$\exists_{People}$ `x Location(x, in, vehicle)`
$\forall_{People}$ `x Location(x, in, vehicle)`

The first example is a context where there is at least one person in the vehicle, while the second one refers to all people in the vehicle.

### Widgets

Widgets are responsible for gathering context information from sensors and generating the facts that reflect the current context. For instance, a widget that provides context information about a person and his location, advertises that it is able to provide the information by means of a first order expression, as shown in listing 3.10:

Listing 3.10: GAIA - Context Example

$\forall_{Person}$ `x` $\forall_{Location}$ `y Location(x, in, y)`

### Interpreters and Aggregators

Interpreters and aggregators get context information from different widgets and by means of rules they derive new context information that can be accessed by context clients. GAIA defines two ways to derive new context information, the first one using rules. Consider again the example presented in section 3.1. In listing 3.11 we specify a rule to deduce that Bob is driving based on existing context.

Listing 3.11: GAIA - Rule Example

```
Vehicle(Engine, is, running) AND
DriverSeat(weight,">", 30)   AND
Location(Bob, in, vehicle)
=> Activity(Bob, is, driving)
```

In listing 3.11 we can see an example of rule that an aggregator would have. However they could contain several rules in order to derive new context. Different rules can become true at the same time, a problem GAIA addresses by providing priorities as a conflict resolution strategies. The second way to derive context information is using machine learning approach using a Bayesian algorithm. This is used to derive for instance user's mood, based on context information such as location, weather, etc.

### Context History

GAIA keeps context information in a database, where context is saved. Context clients are able to reason about past events. Also, saving context information allows GAIA to implement some intelligent behavior for predicting user actions.

### 3.5.2 Context Sharing

#### Context Clients

Like in the previous approaches, applications get context information in two ways. Querying widgets is the first way. For instance we can ask who is in the vehicle: `Location(?x, in, vehicle)`. Variables are represented by a question mark before the name. The evaluation of the query returns all the context that match the previous. The second form is that applications subscribe with the discoverer to receive notifications when the context change.

#### Discoverer

Like in the Context Toolkit, GAIA provides a centralized component where widgets advertise the context information that they provide in the form of first order expressions. Each context client can get a reference to a specific widget through the discoverer.

### 3.5.3 Context Reaction

Context clients must be provided with a mechanism to trigger context changes. GAIA achieves this through the concept of *configuration file*. A configuration file contains a set of context conditions which trigger the associated actions when they become true. These actions are predefined and have to correspond to already implemented methods. Conflict resolution between actions is done using priorities. In listing 3.12 we present an example configuration file for our example described in section 3.1

Listing 3.12: GAIA - Configuration File Example

```
∃Person x Activity(x, is, driving) AND NOT PlayingMusic()
        PlayMusic()
        Priority: 1
Location(Bob, in, vehicle) AND Location(Alice, in, vehicle)
AND PlayingMusic()
        Sound(volume, set, discreet)
        Priority: 2
Locations(Bob, in, vehicle) AND PlayingMusic()
AND NOT Location(Alice, in, vehicle)
        Sound(volume, set, high)
        Priority: 2
```

With this configuration file we first enable that when someone is driving and the music is not playing, then the music player should start playing music. The second and third rule change the volume setting based on the presence of Alice in the vehicle.

### 3.5.4 Conclusions

The use of first order logic allows GAIA to separate the application logic from the adaptation logic and also provides an expressive model for context. However, the configuration file which describes how to adapt according to context must be specified by developers, since the ontology and possible actions of the system are not part of the end-user's knowledge. As a consequence, a limited set of specific scenarios, considered by developers as relevant, are taken into account excluding end-user intervation.

## 3.6 LIME

In this section we present the LIME approach, which constitutes one of the concepts underlying CRIME, which is the framework we used for our experiments. LIME is an approach which is mostly concerned with context sharing in a mobile environment. It uses the concept of tuple spaces and as consequence is a content-based coordination approach [MPR01] for sharing available context information. In LIME, tuple spaces are adapted to achieve coordination among mobile environments where decoupling in time and space are needed.

### 3.6.1 Context Model

LIME [MPR01] does not provide widgets to get sensor information, rather context is represented by tuples. LIME stands for Linda in a Mobile Environment. Linda [Gel85] is the first implementation of the Tuple Space Model. This model enables, asynchronous interprocess communication. In Linda different processes can communicate each other through a tuple space, where data can be access concurrently. Tuples are a series of typed arguments representing the communicated data like ⟨"data", 12, 3.5⟩. Linda has three basic operations. Adding a tuple to the tuple space is done by an *out(t)* operation, where t represents the tuple to be added. Tuples are removed using an *in(p)* operation where p represents a pattern like ⟨"data", ?integer, ?float⟩. Using pattern matching this pattern matches with the previously presented tuple. It is also possible to read a tuple using an *rd(p)* operation. The difference between `in` and `rd` is that the former removes the tuple and the latter just read it.

LIME breaks up the single tuple space of Linda in multiple tuple spaces. It also extends the operations to allow registering reactions to the appearance of tuples in the tuple space.

### 3.6.2 Context Sharing

Each process has its own *interface tuple space*, and different tuples spaces in the same host are merged together in a *host tuple space*. Tuple spaces of hosts that

are in each other's communication range are merged in a so-called *federated tuple space*. Connectivity between devices determines when they share information or not. Thus, the mobility of devices triggers engagement and disengagement of the tuple spaces. The interface tuple space is accessed using the Linda operations presented in section 3.6.1. Engagement is triggered by the arrival of a new mobile unit. As a consequence, the contents of the interface tuple spaces are merged in a single and atomic *step*. Disengagement also relies on tuple location where transiently shared tuple space are separated as if each mobile device were alone. Figure 3.2 illustrates the engagement and disengagement process.

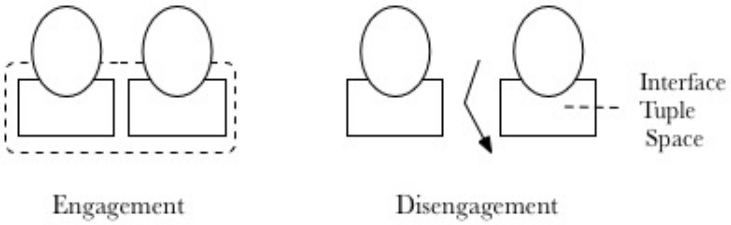


Figure 3.2: LIME - Engagement and Disengagement of tuple spaces

Context clients can access context information using read operations or can receive specific events by means of the weak or strong reactions presented in the next section.

### 3.6.3 Context Reaction

LIME extends Linda by adding the concept of *reaction*. A reaction *R(s,p)* represents a code fragment `s` corresponding to the action(s) that must take place if the pattern `p` matches with a tuple in the tuple space. Reaction can take place one time (i.e when the first matching tuple is encountered) by specifying the `ONCE` mode, while the `ONCEPERTUPLE` mode allows the reaction to take place once per each found tuple.

### 3.6.4 Conclusions

LIME adapts the coordination primitives provided by Linda to the domain of physical and logical mobility. It does not offer a dedicated mechanism for acquired context information and it focuses in the context sharing by means of a distributed tuple spaces. With this approach the problem of a centralized component like the discoverer in Context Toolkit and GAIA is avoided, using the tuple space model context information becomes available at the time devices are connected and merge their tuple spaces. Like in the previous approaches reactions are defined in the based code, using specific scenarios. Furthermore reactions take just one tuple in order to be triggered, not complex description of a composite context.

## 3.7 Conclusions

Applications that dynamically adapt to their changing context require mechanisms to detect and respond to changes in their environment. Context-aware systems focus on facilitating the development of context aware applications for the software developer. They provide different mechanisms from the context acquisition to the reaction in order to adapt to the current behavior. However, incorporating context into applications involves the consideration of a set of concerns more related with end-user needs. As we have seen, dealing with adaptations in these systems is usually heavily tangled with the base code of the applications in the form of if-then predicates. GAIA seems to be a good approach to take the adaptations out of the current code. However like in the other approaches, they need to be completely defined by developers, which may not be completely aware of what the user's needs are. As a result users have context-aware applications only for specific scenarios that may not fullfill their specifications or needs. In this dissertation we propose a methodology to involve the user on context-aware applications. Users must be provided with a mechanism suitable for them. One possibility is using organic user interfaces, which are presented in the next chapter.

# 4
# Context and User Interaction

In this dissertation, we argue for an accessible methodology to involve the user in the development of context-aware applications with an open ontology to describe the world. So far, we have shown that different frameworks have emerged with the purpose to facilitate the development of context-aware applications. They provide an infrastructure for simplifying developer's work. However they are not intended for end-users, which are not involved in the development process. This way, context-aware applications are developed focusing on context acquisition and sharing. Reasoning about context is limited to fit specific scenarios that may or may not satisfy user goals and tasks. User's needs are not expected to evolve since applications do not provide means to support this evolution. Devices may also not change or provide different services or exhibit different characteristics between similar devices, because they are hard coded to specific scenarios. We have identified the need of an interface to involve end-users in the configuration of context-aware applications. This interface should be easy to use, powerful, natural and intuitive. With respect to interfaces that make the human computer interaction (HCI) more natural in a context-aware application, the approach of Organic User Interfaces seems to fit these requirements.

In this chapter, we first present what the principles that underly organic user interfaces. Subsequently, we present some emerging approaches that have been proposed to allow end-users to create and/or configure context-aware applications to fit their own unique needs. Most of them do not require coding, instead they use visual-based interaction which generate code that matches the user's desires. Our analysis in each approach comprises three parts: first of all we analyze what context model is provided for users. Subsequently we describe how the user interacts with the system in order to define the scenarios with the desired behavior. The third part of our analysis highlights how these approaches adhere to the organic user interface principles which are presented in the following section.

## 4.1 Organic User Interfaces

OUIs (Organic User Interfaces) are based on natural laws that relate to physics, biology and human cognition[1]. The process of cognition includes sensation and perception, thinking, imagery, reasoning and problem-solving. OUIs follow four principles: fluidity, intuitiveness, robustness and calmness. These principles aim to improve the usability of user interfaces.

### Principles

*Fluidity* refers to the fact that an interface is prevailed by a simple and clear set of rules. This way, users are guided and restricted by these rules. This principle is based on the observation that all physical phenomena can be explained by means of the laws of physics, without unexpected actions.

*Intuitiveness* aims to give to users a natural understanding of the functionality. In order to do so, it is necessary to understand the human cognition of the world.

OUIs are *robust* since they can survive or recover from some errors.

*Calm* interfaces are non-intrusive. The interaction takes place when users are available. When these interfaces provide information they do not demand that the user stops his current task. But this information is available when the user need it.

## 4.2 CAMP

CAMP (Capture and Access Magnetic Poetry) [THA04] is centered in a home environment that enables end-users to program them. It is based on magnetic poetry metaphor, using natural language. The system uses the W dimmensions [TAB01] to specify specific different scenarios (who, what, where and when).

### 4.2.1 Context Model for Users in CAMP

Context is presented to the user through words in small building blocks, like if they were fridge magnets. CAMP provides a diverse vocabulary. This vocabulary is categorized by color and the W dimmensions presented in [TAB01]. Here we present part of the vocabulary from [THA04]:

who: I, me, everyone, no one, family, stranger, etc.

what: picture, audio, video, conversation, etc.

where: kitchen, living room, home, everywhere, etc.

when: always, later, never, a.m., morning, day, before, Sunday, January, etc.

general: 1, 2, record, remember, view, save, etc.

Considering that a home environment is less dynamic than other environments, this approach will require an extensive vocabulary to enable users to configure their applications in other environments. Even if devices publish their vocabulary

[1] http://media.informatik.rwth-aachen.de/organic.html

to others, users have to check the available services in each environment and define new sentences each time. This could be time consuming and annoying for some users, and there by fails to fulfill the *calm* principle of OUI. One solution to this could be to provide some storage of vocabulary that enable users to define their scenarios later. However since it is just centered in a home environment where devices do not change as dynamically as in others, this is not an issue.

### 4.2.2 User interaction with CAMP

An user study precedes CAMP. The purpose of the study was to understand how users may access and interact with context-aware applications. The results reveal that people do not refer to devices in their scenarios instead they focus on functionality. It is important to notice that this study was performed in a home environment.

In CAMP users describe scenarios with the desired behavior using the available vocabulary for constructing sentences. Furthermore users are able to create new words by combining the existing vocabulary. Users are provided with: a repository which contains all the vocabulary available, a working area where they define their scenarios and an interpretation area that displays the mappings to the specific technology. In figure 4.1 we present an example of a scenario created by a user. This scenario wants to record pictures of baby Billy and display them at the current user's location [THA04].


Figure 4.1: CAMP example of user's scenario [THA04]

After the specification of the scenario, CAMP maps the magnet representation to a specific technology in use. The resulting mapping is displayed to the user, which can then choose to redefine the description. This way users describe their goals without any knowledge of which devices are active in that scenario and what connections are involved. However if we use this kind of approach in an other environment like an office, it may be important to know which devices are available and what properties they have. Consider for instance using a printer service. In an office environment we can have different printers, one black and white printer and one other color printer . In this scenario, users need some means to know which devices are available in order to use one, even if at a first glance they do not care about using the devices.

### 4.2.3 OUI analysis

CAMP is more concerned with the fluidity and intuitiveness of the interface. The rules that dictate how to interact with CAMP are simple and intuitive since users construct sentences representing the desired behavior, like commands. However one problem with the robustness of the system is that it does not provide mechanisms to test if the defined rule is what the user really wants or not. CAMP's interface is calm since users are able to define rules whenever they want, but this is limited to a home environment. An important point is that when a defined rule did not precisely capture what the user wants, it may not be possible to adapt the rule accordingly.

### 4.2.4 Conclusions

Providing a set of atomic functionalities seems to be a very useful way to keep systems easy to use, however the flexibility is limited. Categorization of the context blocks and color coding are mechanisms that enable to group context information and identify them. However this classification is done by developers. Users specify goals in sentences without any explicit knowledge of what devices are acting. From this approach we note the necessity for some kind of context storage that enables the users to define preferences or scenarios at the moment they want it (i.e. when that context may no longer be active)

## 4.3 ACCORD

ACCORD (Administering Connected Co-Operative Residential Domains) uses the metaphor that devices and sensors are represented by puzzle shaped icons [HCH+03]. Like CAMP, it is centered in domestic environments. Since it is more concerned with the ease of use, the level of freedom is limited.

### 4.3.1 Context Model for Users in ACCORD

We can see that it differs with CAMP since the pieces of puzzle represent actual sensor and devices, and in CAMP users do not care about which actual devices correspond to the word in their vocabulary. The reason of using the puzzle pieces to represent the context is in order to control meaningful connections between devices. This way having icons with different shapes users can determine whether a component provides, transforms or uses data as is shown in the figure 4.2.

For instance, a motion sensor is naturally not able to process any input and only provides sensor data. The other way around, a device that provides a certain service like a device that makes sound available does not provide any output that could be meaningfully connected to any other component. However in a more general scenario it would depend on the kind of situation. Since the iconic representation

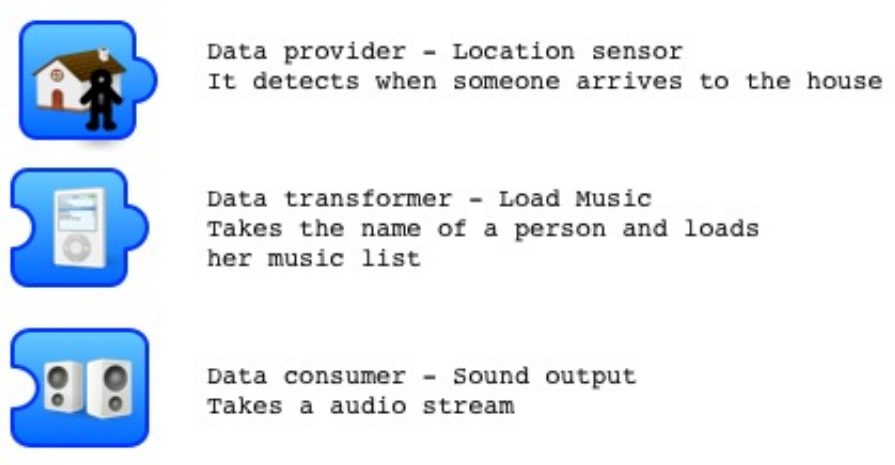


Figure 4.2: ACCORD examples

is given by developers, it can be confusing for users as they can have a different interpretation for them.

### 4.3.2 User interaction with ACCORD

The scenarios are accomplished using the special editor that ACCORD provides. The editor is composed of a repository where every discovered service is made available as a puzzle piece. In the editor the workspace area enables users to connect devices and services to fit their needs using the available puzzle pieces in the repository. Figure 4.3 depicts an example of one scenario defined by a user. This scenario combines the doorbell, the webcam and a portable display. The interpretation of this is that whenever the doorbell sounds take a picture of the person that activates it and show the picture in the the portable display.

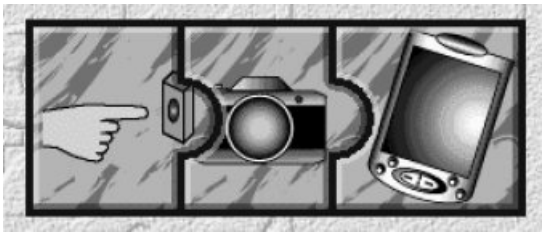

Figure 4.3: ACCORD user's scenario [HCH+03]

### 4.3.3 OUI analysis

ACCORD like CAMP is centered in fluidity and intuitiveness since the metaphor used for the context it is more familiar and easy for users. Fluidity in terms of the rules that restrict the system are quite easily understood by users by using the context representation as iconic puzzle pieces. For users the different shapes representation helps them to intuitively connect different devices. Iconic representation may helps in the intuitively of the context, however since they are not defined by users can become a resource of ambiguity.

### 4.3.4 Conclusions

In ACCORD users make use of predefined functionality. It shows the available services and devices. The applied metaphor is user-friendly but it can be lead to ambiguities in the interpretation. Like CAMP, ACCORD provides a repository of available devices, and a working area to define the scenarios.

## 4.4 CADEL

CADEL (Context-Aware rule Description Language) is a language which has a syntax similar to natural language [NYS+05]. However it also provides a graphical interface where users define the situation with the corresponding action that may take place. Like the previous approaches it is used in a home environment. CADEL focuses on the problem that different users may control the behavior of the same device, leading to conflicting directives.

### 4.4.1 Context Model for Users in CADEL

CADEL uses the rule approach, where we have some conditions that have to be met in order to trigger an action. It also follows a syntax similar to natural language. Thus scenarios can be described like: At night, if the entrance door is unlocked for 1 hour, turn on the alarm [NYS+05]. Also, it enables to define new words based on others that already existed. However the natural language may not be user-friendly enough, because users have to memorize the specific syntax. CADEL provides a graphical interface in order to assist the user and make the rule creation more intuitive.

### 4.4.2 User Interaction with CADEL

Figure 4.4 presents the provided interface. CADEL's interface like ACCORD's shows the available sensors and devices. CADEL also enables users to define new words.

As we already mentioned, CADEL provides a mechanism to avoid conflicts between different rules that affect or change the state of the same device. Since

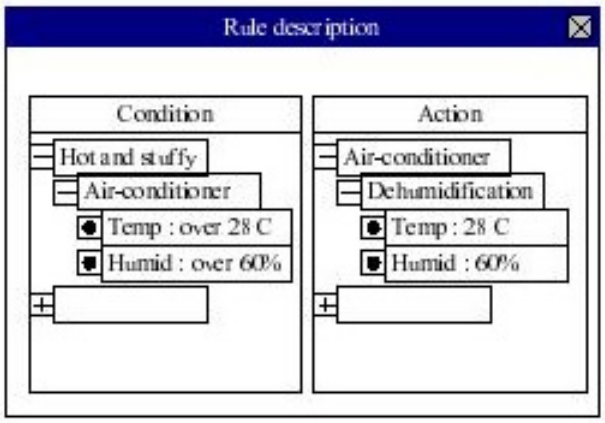


Figure 4.4: CADEL Interface [NYS+05]

context is subjective to the user, different users may expect different behavior from the same device. For instance, one user prefers a temperature that for others may be too cold or too warm. Thus users use an interface to specify which rule has higher priority.

### 4.4.3 OUI analysis

CADEL is an intuitive interface since it follows the if-then rules definition. Like in the previous approach users has to define rules when the services are available. This is due to the fact that devices and services that are currently available offer to build new rules.

### 4.4.4 Conclusions

CADEL provides a means to users to look up devices and services. It assists users in the rule definition that dictates the behavior in according to particular circumstances. Each user has different preferences and perceives their environment differently. A priority mechanism is provided in order to resolve conflicts between different rules that affect the same device.

## 4.5 CAPpella

CAPpella is a programming by demonstration Context-Aware Prototyping environment intended for end-users [DHB+04]. Programming by demonstration means that the user interacts with the system to show how the desired program should work.

### 4.5.1 Context Model for Users in CAPpella

The environment is captured and represented to the user using data streams in a graphical sheet. This graphical sheet contains all the output of the sensors and the time. An example of this representation is presented in figure 4.5 in the sampling sheet part.

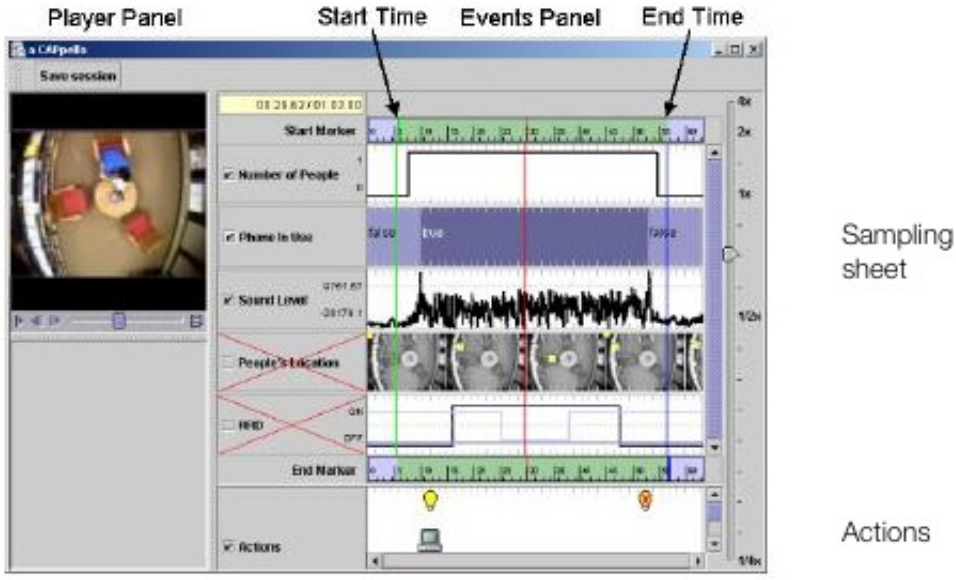


Figure 4.5: CAPpella Interface [DHB+04]

### 4.5.2 User interaction with CAPpella

CAPpella provides a recording system: users are only responsible to train the application. After the user starts the recording system, CAPpella captures the available data from the different sensors, and also keeps track of all actions that take place during the recording. During this recording time, users show how the application should behave and execute the actions. Afterwards, users select the stream information they consider to be relevant. Users may train the application until they are sure that it performs the correct actions in the situations that they are interested in. Thus a kind of learning process takes place while training. Users are able to discard samples of data that may not be significant for the training of the system.

### 4.5.3 OUI analysis

Even though the context model for the user is not as intuitive as CAMP or ACCORD, the interaction model allows for fluidity. However users may feel annoyed

during the training period. The use of low-level information in the interaction interface can be regarded as unintituive since users do not care about low-level information. The programming by demonstration makes the system intuitive, since users are interacting directly with the context.

### 4.5.4 Conclusions

Users are required to work directly with sensor output to select streams which they consider to be relevant. We consider this a drawback since users may not have knowledge about what the relevant streams are for the current setting. Users are forced in some way to deal with low level information. Machine learning is one of underlying the principles of this approach. By means of identifying actions and situations represented by sensor streams the user can effectively train applications.

## 4.6 iCAP

ICAP (Context-aware Application Prototyper) [DSSK06] enables the creation of prototype context-aware applications without writing any code. It allows a higher degree of flexibility, but as a consequence it introduces more complexity. Like CADEL users specify the situation and the associate action in a rule-based approach.

### 4.6.1 Context Model for Users in iCAP

Users are responsible for creating the entities of the environment and the actions. Users may create each element they will use and associate an iconic representation for each. Thus users work with icons that they associate, and that are significant for them.

The rules that guide the behavior are categorized: activity, object, location, time, person and state. People entities have preferences that enable them to configure devices for their personalization environment. These preferences include light, sound and temperature. Users can specify if their personalization preferences should be activated in a specific location or not.

This approach emphasized the use of relationships, such as spatial relations between different locations or social relations betwen persons (family, superior, etc.)

### 4.6.2 User interaction with iCAP

Like CAMP a user study was done. Their results show that users describe context-aware applications in terms of if-then rules. And in contrast with CAMP, they observed that users were focused on objects. However users are not concerned with sensors or sensing.

Thus the user interaction with iCAP is described by if-then rules. Entities are created by the user: she has to determine which entities are relevant, create them if they do not already exist in the repository. For each entity users add an icon that enables to personalize and have a more friendly representation. Also, each entity is associated with a group (e.g colleague, family) which was previously defined by the user. Such groups allow categorizing entities, and it is a means to establish relations between entities.

Users also create the outputs and also associate a category to them: an iconic representation and the type of output (e.g. binary to specify on/off).

The iconic representation then enables to create the rules by dragging and dropping them to the rule sheet. The rule sheet consists of two parts: the condition sheet and the action that takes place when the conditions are met. In figure 4.6 we can see both parts of the interface.

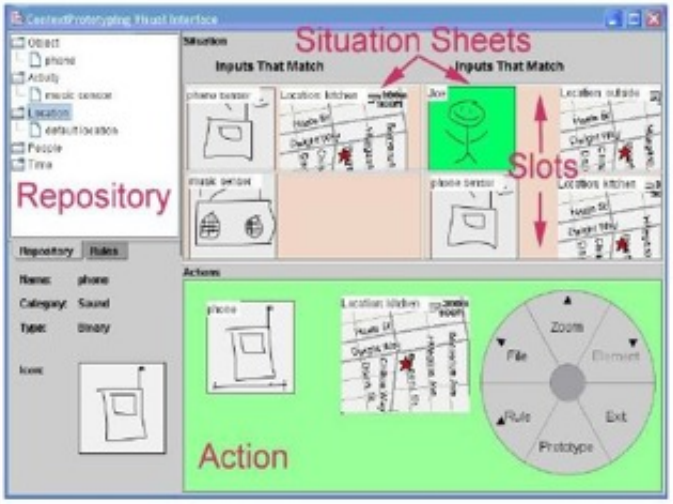


Figure 4.6: iCAP Interface [DSSK06]

With iCAP users are able to prototype, test and deploy context-aware applications. In order to test them it provides an interface that enables to simulate the values. Figure 4.7 displays this interface.

### 4.6.3 OUI analysis

In terms of fluidity and intuitiveness, iCAP exhibits quite complex rules. However, they give more freedom to use it in more general environments. Different aspects make iCAP intuitive. First of all, its metaphor is based on a user study that shows that users perceive context-aware scenarios as if-then rules. Furthermore the use of iconic representations given by users enable that in some way they describe context

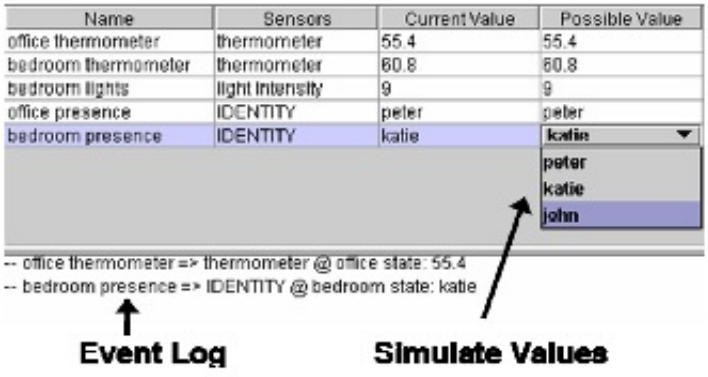


Figure 4.7: iCAP Test Interface[DSSK06]

as they conceive it. Also it enables creating connections between entities by means of the different relationships that can be established. Providing the testing of the rules, users make sure that there will not be undesirable behavior in their devices.

### 4.6.4 Conclusions

iCAP constitutes a rich environment to involve users in the prototyping of context-aware applications. The concept of introducing rules and restricting the choice is useful in the customization of the context-aware applications. Categorizing and relating entities seams to be a natural way of user cognition of the world.

## 4.7 Conclusions

In this chapter we have reaffirmed that users must be provided with mechanisms to express their needs. The use of these mechanisms should not require specialized technical knowledge in order to extend the power of context-aware applications. These approaches show and recognize that end-users need a simple way to specify behavior for their applications. As users and their environment evolve, context-aware applications need to provide the mechanisms for coping with such evolution. They offer different mechanisms and languages that require little or no programming knowledge.

We identified important mechanisms that could be provided for end users in order to give them an interface that enables the configuration and use of context-aware applications. Almost all approaches shown provide a repository with services or devices available. This repository allows users to create their own scenarios using the services or devices available. Categorization and mechanisms to establish relationships are important. Also we identify the need to provide mechanisms to define their rules at the moment they desire to do so.

# 5 Crime

In this dissertation, we argue for an accessible methodology to involve the user in the development of context-aware applications with an open ontology to describe the world. Having explained the mechanisms to develop context-aware applications, this chapter focusses on the CRIME coordination language which is the basis of our experiments described in the next chapter.

CRIME (Consistent Reasoning In a Mobile Environment) is a logic coordination language for mobile ad hoc networks which offers a structured way to deal with the device disconnections [MSP+07]. This is done by asserting and retracting facts to constantly reflect the current context and keeping a link between causes and consequences in rules.

This chapter comprises two main parts: first of all the CRIME coordination language itself is introduced. Subsequently we describe the fact space model which captures the distribution model underlying the CRIME middleware.

## 5.1 Logic Coordination Language

The Logic Coordination Language enables applications to reason about their context. An application is able to adapt to the current state of its environment by means of rules defined in this language. The logic coordination language is based on the logic programming paradigm which we presented in chapter 2.

The Logic Coordination Language is a logic language which is interpreted by means of forward chaining (as more specifically Rete). In order to more efficiently support retractions and reassertions of facts, CRIME implements the Rete algorithm detailed in chapter 2. Since CRIME is based on logic programming, it has facts and rules. The CRIME syntax resembles Prolog syntax.

### 5.1.1 Crime Syntax

In CRIME we can have literals like numbers, strings and symbols. Variables are preceded by a question mark.

#### Facts

Consider Bob's mobile phone. It has a fact encoding his personal information:

Listing 5.1: CRIME's Fact

```
userInfo(925251, Bob, Ross, "bob_ross@vub.ac.be", VUB).
```

A fact in Crime is formed by a type and a series of arguments. It may not contain any variables. In this example, the `userInfo` is the type of the fact. The fact contains: an user's id, the name, the family name, his email-adress, and the place where he works. Note that a fact can only contain literals (numbers, strings or symbols).

#### Rules

Rules follow the `action` ← `condition` form. In Crime a rule has prerequisites and consequences. The prerequisites of a rule are formed by facts that can contain symbols and variables. Variables in Crime are denoted by a question mark preceding their name. The consequences can be application/user specific actions or facts. The application or user specific actions are denoted by a colon prefix. These kind of actions are Java classes that inherit from the Action class. Crime uses the symbol ":-" to separate between prerequisites and consequences and it can be interpreted as an "if". Like in Prolog, a comma is used as an "and" operation between the prerequisites.

Consider the following mobile phone example. This mobile phone is equipped with an accelerometer. This sensor makes it possible to detect when the mobile phone is rotated from portrait to landscape and viceversa. A fact representing the current position of the cell phone may look like: `position(portrait).` Consider for example the following rule:

Listing 5.2: CRIME Display Configuration Rule

```
:changeDisplayTo(?application, ?currentPosition) :-
        position(?currentPosition),
        running(?application).
```

In this rule, we have an action which is called `ChangeDisplayTo` whose variables should be bound in the prerequisites. This rule enables changing the current display configuration for the application currently running. Crime also enables the use of other constructs from Prolog like `not`, and accumulating constructs like: `bagof`, `findall` and `length`.

**Not** Coming back to the example presented in section 3.1, if we want to define the volume profile then we must have a rule like the one in listing 5.3

Listing 5.3: CRIME Rule using Not

```
:ChangeSoundProfile(Loud), profile(Loud) :-
        activity(925251, driving),
        colleague(Bob, ?colleague),
        not person(?colleague).
```

With this rule we change the sound profile if there is no colleague and Bob is driving. The negation consist in just add the not to the prerequisite.

**Findall** Consider that we want to know all the persons that are in the vehicle. We must define a rule like the one presented in listing 5.4. The `findall` statement has three arguments: the first one (e.g. `?person`) represent a variable which occurrence will be checked in the query of the second argument (e.g. `location(?person, vehicle)`), and the third argument is the variable where all the occurrences in the query are accumulated in a list (e.g. `?persons`).

Listing 5.4: CRIME Rule using Findall

```
present(?persons) :-
        findall(?person,
        (location(?person, vehicle). ),
        ?persons).
```

Thus if we have facts presented in listing 5.5, with the rule of listing 5.4, the fact `present([925251, 919212])` will be asserted.

Listing 5.5: CRIME's Facts of location

```
location(925251, vehicle).
location(908070, house).
location(919212, vehicle).
```

**Bagof** The `bagof` statement is slightly different from `findall`, consider the rule defined in listing 5.6. The `bagof` statement allows to accumulate according to other variables. Consider again the facts in 5.5. with the rule of listing 5.6 we will assert the two following facts: `present([925251, 919212], vehicle)` and `present([908070], house).`

Listing 5.6: CRIME Rule using Bagof

```
present(?persons, ?place) :-
        bagof(?person,
        (location(?person, ?place). ),
        ?persons).
```

**Length** On the other hand, the `length` statement can be used with `findall` and `bagof`, the `length` statement has two arguments one that is a variable which contains all the accumulated values in a list and another that corresponds to the number of elements in the list. Thus for example the rule presented in 5.7 will assert the fact `number_present(2).` Hence this rule gets how many people there are in the vehicle.

Listing 5.7: CRIME Rule using Length

```
number_present(?number) :-
        findall(?person,
        (location(?person, vehicle). ),
        ?persons),
        length(?persons, ?number).
```

## 5.2 The Fact Space Model

The fact space model allows the coordination between applications and it gives them a consistent view of their environment [MSP+07]. With the fact space model, applications perceive their environment as a set of facts constituting the knowledge base in a federated space, instead of tuples like in LIME presented in chapter 3

In order to provide the necessary support to exchange context information in a mobile environment, this model provides an application with multiple fact spaces. At least each application has a private interface fact space whose facts are not shared, whereas the facts residing in other interface spaces are published or shared. The interface fact spaces are integrated with the host level fact space which enables them to transparently share context information between applications in the same host. Figure 5.1 illustrates such host level fact spaces on two disconnected host for our scenario presented in section 3.1.

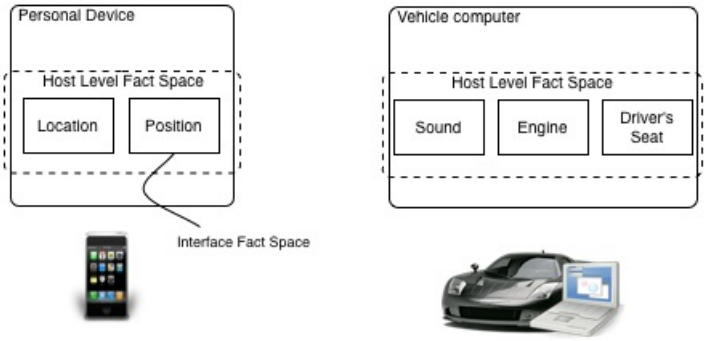


Figure 5.1: Fact Space Model on disconnected devices

### 5.2.1 Federated Fact Space

When different hosts get into one another's connection range, their host level fact spaces are joined into the federated fact space as is illustrated in figure 5.2.

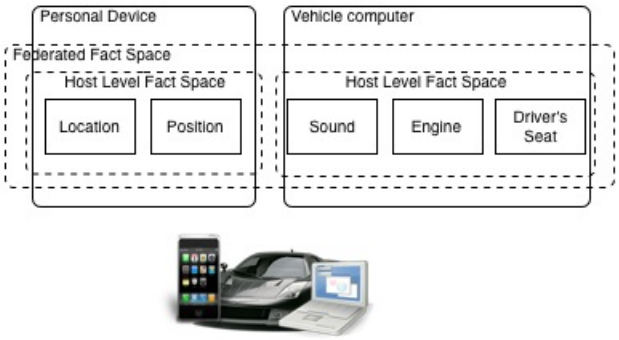


Figure 5.2: Fact Space Model - Federated Fact Space

Applications in the fact space model have the option to adapt themselves when they assert the shared facts from other applications in the exchanging process. When the facts representing the current context are asserted, new facts can be asserted or some actions take place in order to adapt the application. The retraction of facts takes place when the host that shared them it is no longer in the connection range. This model enforce, having a compensating action that takes place when facts are retracted.

#### Quantified facts

In order to implement the federated fact space model, Crime provides quantified facts, they have the following form:

Listing 5.8: CRIME - Quantified fact

```
public -> location(925251,vehicle).
```

In this example the location fact is `public`, hence it is shared. The fact space can take any name. If a fact does not have a fact space specified, then it is considered to be `private` and it will not be shared with other applications in the current devices and other applications running on co-located devices. Likewise, it is possible to define rules based on the fact space of facts or define actions that assert facts in a specific fact space.

### Crime Actions

In Crime, actions must inherit from the `Action` class. Thus inheriting from Action class enforces to have two specific methods: `activate` which is executed when all the prerequisites are met; and the other method is `deactivate` which describes a compensating action that takes place when at least one or more prerequisites are no longer met.

Consider the example presented in section 3.1, where Bob wants to change his profile, to redirect all the calls to the answering machine when he is in his vehicle. To do so, he has defined some preference facts like the following:

```
answerMachinePreference(vehicle, on).
```

Also, he has some part of his personal information available:

```
public -> userInfo(925251, Bob).
```

Here, we have that Bob's Pda has defined the following rule:

Listing 5.9: CRIME Rule to redirect the sound of a Pda

```
:answerMachine(?state) :-
  public -> location(myID, ?place),
  answerMachinePreference(?place, ?state).
```

This rule means that if Bob has a call preference where he specifies the place `answerMachinePreference(?place, ?state)` and the place is equal to his current location `location(myID, ?place)` it may change the answer machine state. Thus answerMachine action may look like:

Listing 5.10: CRIME action

```
public class AnswerMachine extends Action {
  public static Pda pda;
  public void activated (Vector args) {
      pda.answerMachine(args.elementAt(AttributeValue.STATE));
  }
  public void deactivate (Vector args) {
      pda.answerMachine(AnswerMachine.DEFAULT) ;
  }
}
```

So, when Bob get in his vehicle, the vehicle detects his presence by means of a computer in the vehicle. Bob's Pda and the computer in the vehicle are co-located and exchange context information. Bob's Pda shares the `userInfo`, and based on this the vehicle's computer publishes a fact indicating that the person with user identifier 925251 is present:

```
public -> (location(925251,vehicle)).
```

Since the prerequisites of the answerMachine rule are now met, it is activated and the answering machine is turned on. When Bob gets out of his vehicle, the fact:

```
public -> (location(925251,vehicle)).
```

will be retracted. As much the prerequisites are no longer all met, so the `deactivate` method is executed, resetting the answering machine to the default value.

## 5.3 Conclusions

The Fact Space Model used by CRIME is able to manage the disconnection in a ad hoc network. This is done by keeping a link between causes and consequences, in order to have a compensation action when the disconnection takes place. The context information, represented as facts, is asserted to the knowledge base when it is available and retracted when it is not. Also this information is shared in the host and federated fact space. The assertion and retraction of facts gives applications a consistent view of their environment, enabling them to adapt to them.

We have presented the principles of CRIME, since it constitutes the basis for our experiments presented in the following chapters.

# 6 An Open Ontology in CRIME

In this dissertation, we argue for an accessible methodology to involve the user in the development of context-aware applications with an open ontology to describe the world. So far we have explained the traditional mechanism to develop context-aware applications where users are not involved in the configuration process that determines which environmental aspects are important in order to perform adaptations. Thereafter, we presented some approaches that try to involve the user in configuration or prototyping of context-aware applications, using GUI interfaces. We also described CRIME, which is the basis for our experiments. In this chapter we focus on some extensions we did to CRIME to make it easier for the user to describe the world. We make a extension to the CRIME context model, which allows to establish hierarchical relationships and enable to make partial matching.

## 6.1 Describing the World

When we are in the field of context-aware applications we have to consider highly dynamic environments and heterogeneous real world situations. These scenarios involve many types of entities, where each entity may offer different services and exhibit different characteristics. All these things must be modeled and handled in a context-aware setting. Consider the following situation:

*Bob constantly makes use of different printing services. At work, he just prints papers and reports, thus he just makes use of the black printer. However, Bob works with variable groups of people in different floors of his building. Each floor offers different printing services. Each floor is equipped with at least one black printer and one color printer. At home, sometimes he prints his photos or images with his color printer. Also, he prints interesting articles with his black printer.*

In this scenario we are faced with different entities that provide the same service of printing. However each entity may have different characteristics (e.g. black

or color), and may or may not exhibit different characteristics.

### 6.1.1 Describing the World in CRIME

Currently in CRIME, context information is represented as facts. Each fact is composed from attributes that must be constants. In addition, CRIME enables to reason about the context using rules. Rules formed by actions and conditions. These conditions are formed by patterns where the main difference with facts is that they can contain variables. Reviewing the presented scenario we can have facts like the ones depicted in listing 6.1, where the semantics of the attributes are preconceived by the developer.

Listing 6.1: CRIME - Facts without data structures

```
printer(P10CAB, black, laser, 600, VUB)
printer(P98PO, color, inkJet, 500, VUB)
```

Consider that Bob makes two different rules. When he is working with documents and he wants to print them, he wants to send them to a black printer, while for printing images he uses a color printer, as it is illustrated in figure 6.1.

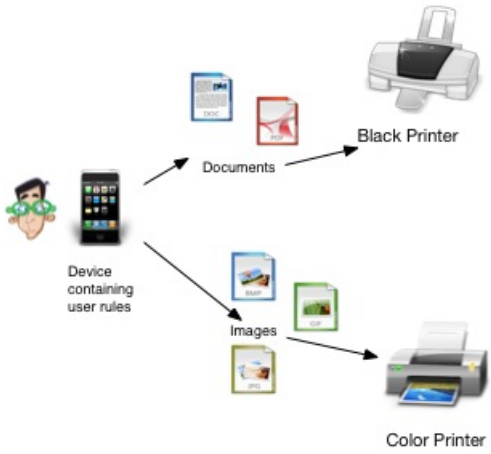


Figure 6.1: Scenario

Thus in order to accomplish the specified behavior, Bob can define rules like the ones presented in listing 6.2.

Listing 6.2: CRIME - Rules without data structures

```
:Print(?printer, ?fileName) :-
    printer(?printer, black, ?type, ?dpi, ?brand),
    print(?fileName, document, ?size, ?creator, ?date).

:Print(?printer, ?fileName) :-
```

```
    printer(?printer, color, ?type, ?dpi, ?brand),
    print(?fileName, image, ?size, ?creator, ?date).
```

Some problems can arise with the current facts and rules. First, the *semantic* of each attribute in a fact is not clear. As a consequence, in order to define rules about it, there must be some mechanism to know the real meaning of each attribute. Also, *positional* dependencies, for example each printer exhibit their id, if it is color or black, its type, its resolution and its brand, this characteristics must be in the given order, if one printer has other order of attributes like having its type before the color or black type, for instance: `printer(P98PO, inkJet, color, 500, VUB)`, the second rule is no valid for this printer, and it will be necessary to define a new rule even if the functionality is the same, just because of positional restrictions. Other problem that we identify is an *arity* constraint: we can see that each entity must exhibit the same number of characteristics in order to define a general rule. Thus if Bob has at home a printer that exhibits the following: `printer(P40PH, color, laser, 800, EMOOSE, photo)`, in this fact a new attribute is added (photo) which corresponds to the quality. So, Bob has to define other new rule like the one presented in listing 6.3 to consider also this printer. In previous examples, we expect that printer facts contain five attributes instead of six, this small difference makes the rule presented in listing 6.2 not valid for these kind of facts, even if the other attributes are the same. Given that the number of attributes and the order of them are important, we have to add variables to fulfil all the attributes even if we do not use them as parameters in the action. In our examples we have to add the printer's brand and the file's creator as variables even if they do not constituted relevant information for the rule definition.

Listing 6.3: CRIME - Other Rule without data structures

```
:Print(?printer, ?fileName) :-
    printer(?printer, ?type, color, ?dpi, ?brand, ?quality),
    print(?fileName, image, ?size, ?creator, ?date).
```

## 6.2 Composition and Structuring Mechanisms

Composition is a rich mechanism that enables to have one piece of data composed of other pieces, that in some cases may be composed from other pieces as well. In figure 6.2 we present how the composition information occurs by defining userInfo, consider that each circle it is a piece of data describing the world. However as we can see, some relations, that establish some structure in the information, must exist between them and some times they exhibit more characteristics than others given the size of it. However it is also possible to create hierarchical relationships between them.

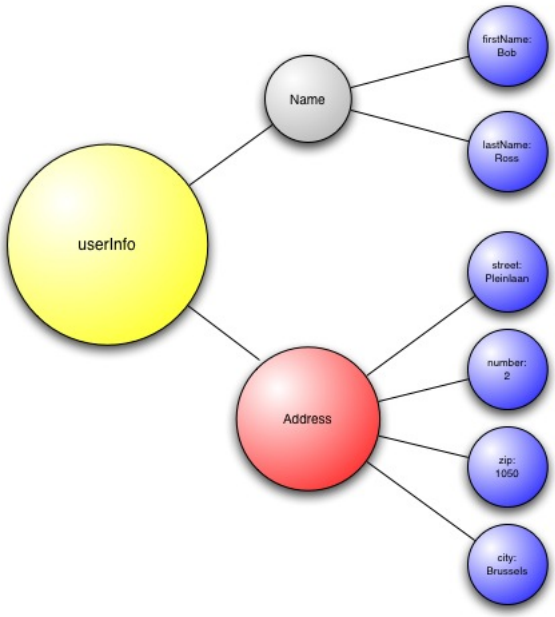


Figure 6.2: Composition and Structuring of contextual data

### 6.2.1 Lists

Lists are useful data structures. We have added lists to CRIME using the Prolog syntax. We enable two types of syntax to express lists. Lists are enclosed between square brackets. An example of a common list is `[elephant, dog, cat, frog]`. They enable to group information that is related in some way. They can contain any attribute as list element, even other lists.

If a bar "|" is put just before the last term in a list, it means that this last term denotes a sub-list. Inserting the elements before the bar at the beginning of the sub-list yields the entire list. For example, `[elephant, dog, cat, frog]` is the same as `[elephant, dog | [ cat, frog ] ]`, and also the same as `[elephant | [dog, cat, frog ] ]`

Reviewing the example of the printing services, using the list in CRIME, facts may look like the ones presented in listing 6.4.

Listing 6.4: CRIME - Facts using Lists

```
printer([P10AB, black, laser, 600, VUB])
printer([P98PO, color, inkJet, 500, VUB])
printer([P40PH, color, laser, 800, EMOOSE, photo])
```

In listing 6.5 we define again the same rules but now using lists.

Listing 6.5: CRIME - Rules using Lists

```
:Print(?printer, ?fileName) :-
```

```
    printer([?printer, black| ?detail]),
    print([?fileName, document| ?rest]).

:Print(?printer, ?fileName) :-
    printer([?printer, color| ?detail]),
    print([?fileName, image| ?rest]).
```

We review the problems presented in the previous section (semantic ambiguity, positional dependencies, arity limitation, unused variables) but now with the use of lists. First the *semantic* of each attribute in the fact is still not clear. However it is possible to group information using a list and to make some hierarchical relations since lists may contain list as well. *Positional* dependencies, are still present since we have to know the position of the attribute of interest. However the *arity* limitation is somehow solved since we only need to know the number of attributes before the one that is of our interest. Thus for the case that Bob has in his house a printer that exhibits the following where we can identify that the list has one element more `printer([P40PH, color, laser, 800, EMOOSE, photo])`. This way the rules presented in the listing 6.5 are still applicable, since the fact `printer([P40PH, color, laser, 800, EMOOSE, photo])` matches with `printer([?printer, color| ?detail])`, even if it includes the quality attribute (photo quality) that other printer's facts do not have. Unused variables depend of the position in the list of the interested attribute if the attribute that we want to use to filter or that it is a parameter in the action is the last one, then we must specify all the attributes. For instance, if we want to print just in printers with brand equal to VUB, we already know that the brand is the fifth attribute, thus we have to add the first four attributes too and it will result in a pattern like `printer([?printer, color, ?type, ?resolution, VUB | ?detail])` in the condition part of the rule. However if the attribute that we are interested is the first one, we just need to get the head of the list and the rest is irrelevant for us, taking out some unnecessary variables.

### 6.2.2 Records

Given that the lists still present some problems with the flexibility of the world representation, we propose records. To introduce records in CRIME we add the JSON syntax [Cro06]. Adding these records to CRIME was made by adding a new kind of attribute, in next section we give more details on this.

#### JSON syntax in CRIME

JSON (JavaScript Object Notation) is a lightweight data-interchange format [Cro06]. It is language independent. JSON is built using records which consist of a collection of name/value pairs. This notation enables to specify significant meaning to each attribute and interrelations between them.

Using JSON records we balance between a rigorous specification and ease of

use. This balance is achieved by using a pair relation that enables to specify the vocabulary, which meaning is unambiguous. Thus developers are responsible to define meaningful semantic relationships between the terms.

A record is an unordered set of name/value pairs. A record begins with "{" (left brace) and ends with "}" (right brace). Each name is followed by ":" and the name/value pairs are separated by ";". Also we enable the different relational operators, not only equality.

Consider again the example so far presented in this chapter. We can have facts like the ones shown in listing 6.6.

Listing 6.6: CRIME - Facts using Records

```
printer({ id:P10AB;
          prints:color;
          type:laser;
          resolution:600;
          brand:VUB }).
printer({ id:P98PO;
          prints:color;
          type:inkJet;
          resolution:500;
          brand:VUB }).
```

Also we can make more generalized facts like the ones presented in figure 6.7.

Listing 6.7: CRIME - Facts using Records

```
service( { printer:
                { id : P10AB;
                  type : laser;
                  prints : color;
                  resolution : 600;
                  brand : VUB }
         } )
```

In listing 6.8 we define the same rules we have been defining so far, but using records. Going back to the analysis we have been doing, we can see that the *semantic* of each attribute now is clear. It is a fact that it also depends on the name given to each attribute. The positional problem is solved since now it only depends in the name given to the attribute. The rules are applicable to all the facts of the same type that have the attributes given. We are not longer dependable of the number of attributes that one printer has an other not. We only depend of the type of fact and that it exhibit the same name attributes that we are interesting in. We make *partial matching* since we just match the attributes that we are interested in. Thus we also avoid adding unnecessary variables, since we only specify the ones that are of our interest. We can make finer control, since we can add relational operators like in the second rule of listing 6.8; here we do not just filter color printers but we are also interested in printers which resolution is greater or equal than 600.

Listing 6.8: CRIME - Rules using Records First Form

```
:Print(?printer, ?fileName) :-
        service( { printer:
                        { prints : black }
                 } ),
        print( { file : ?fileName;
                 type : document } ).

:Print(?printer, ?fileName) :-
        service( { printer:
                        { prints : color;
                          resolution >: 600 }
                 } ),
        print( { file : ?fileName;
                 type : images } ).
```

We enable two kinds of notations using the records. In listing 6.9 we present the second notation which meaning is the same as the listing 6.8. In this notation we can bind variables with attributes and then go inside to the attributes of the bounded variable. If we look for a specific attribute using the variable, this variable must first be bound with a record, otherwise will not match. This notation enables to use the complete records, and also pieces of them. For example if we want to use complete file information, we just make use of the `?file` variable, but if we need some particular information, we just take the part we are interested in, like `?file.filename : ?filename`, to get file's name.

Listing 6.9: CRIME - Rules using Records Second Form

```
:Print(?printer, ?fileName) :-
        service(?service),
        ?service.printer : ?printer,
        ?printer.prints  :  black,
        print(?file),
        ?file.filename : ?filename,
        ?file.type : document.

:Print(?printer, ?fileName) :-
        service(?service),
        ?service.printer : ?printer,
        ?printer.prints  :  color,
        ?printer.resolution >: 600,
        print(?file),
        ?file.filename : ?filename,
        ?file.type : document.
```

## 6.3 Rete Extension

Some extensions to the Rete algorithm were necessary to support list data structures, records and the use of relational operators in CRIME. In addition we have also added three new constructs that enable: create persistent facts that will help to create a history of context information, a predicate that enables to get all the current context available and a history one to get the past context.

### 6.3.1 Parsing

The parsing process of a rule is the first step to get the Rete network. In [SP07] the Backus-Naur form of a simplified grammar of CRIME is presented. We present in listing 6.10 the extensions that we did in order to support more complex data structures, presented in the previous section.

Listing 6.10: Syntax of the CRIME extensions in Backus-Naur form

```
<rule>       ::= <actions> ':-' <conditions>
<actions>    ::= <action> ',' <actions> | <action>
<action>     ::= <ser action> | <pattern>
<ser action> ::= ':' <pattern>
<conditions> ::= <pattern> ',' <conditions>
<patterns>   ::= <pattern> ',' <patterns> | <pattern>
<pattern>    ::= <type> '(' <attributes> ')'
<attributes> ::= <attribute> ',' <attributes> | <attribute>
<attribute>  ::= <variable> | <cte> | <list> | <record>
<list>       ::= '[' <attribute> | <attribute> ']'
                 | '[' <attributes> ']'
<record>     ::= '{' <pairs> '}'
<pairs>      ::= <pair> ';' <pairs> | <pair>
<pair>       ::= <key> <relational_operator> <value>
```

We have seen that each pattern, that forms the condition of a rule, has attributes. Our extension proposes to add new kinds of attributes like lists and records. We already mentioned that each list can take the form of head and tail. It can also take the simple form of a list of attributes, each one follow by a comma. However our internal representation follows the head-tail notation. On the other hand, records consist of key-value pairs. In the condition part of a rule we can have records that enable to filter using relational operators, not just equality. We include operators like "!:" not equal to, "<:" less than or equal to, ">:" greater than or equal to, "<" less than, ">" greater than.

In figure 6.3 we present a diagram of some of the extensions we did. We extended the `Attribute` class with a `CompoundAttribute` class which enables to have an attribute composed of other attributes.

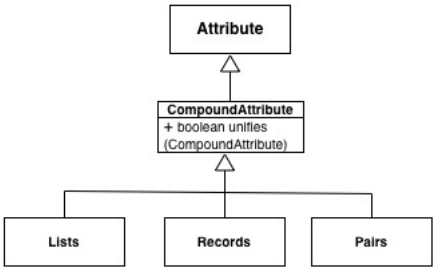


Figure 6.3: Attribute Extension

### 6.3.2 Filter Nodes

Filter nodes constitute the alpha network in the Rete algorithm. We had to extend filter nodes to provide support for this kind of structures. The `FilterCompound Attribute` class is derived from the abstract class `Filter` where we define the pass method. We use indexing, this index represent the complete address of an attribute, and recursively matching between all parts of the compound attribute. In figure 6.4 we present a detailed index representation for the `userInfo` example using list, where we just specify the name composed of the firstname, the family name, and the age. Also, in figure 6.5 we present the same example, but using records. Filter nodes, filter attributes based on this indexing. Filtering a compound attribute includes all the constant filters that form the complex data structure. For example in the case of lists, consider that we have a prerequisite with the following pattern: person([[Bob|?lastname]|?age). For this pattern one filter node is created to filter those facts which contains Bob in the correct position. This filter node contains the exact address (0 → List.HEAD → List.HEAD) where it has to be the Bob symbol. If it is not possible to reach the address or the value found is not equal, then it does not pass the filter and no token is passed. In the case of records it is the same approach, using the index. Additionally, in the case of records, checks are performed in terms of relational operators.

### 6.3.3 Join Nodes

Join Nodes make use of the `VarChecker` class in order to check between different variables. However an extension was required in order to support not only the equality checks between variables, but also checks using different relational operators. In order to support relational operators a variable checker must include the comparator that will be used to perform the check. This parameter is additional to the four indices indicating the index of the fact in the right and the other in the left side of the join node, the index of the attributes that must be checked. Nevertheless each variable can be put under different constraints. Consider for example that we want to find people older than Bob. We will have a rule like the one presented in

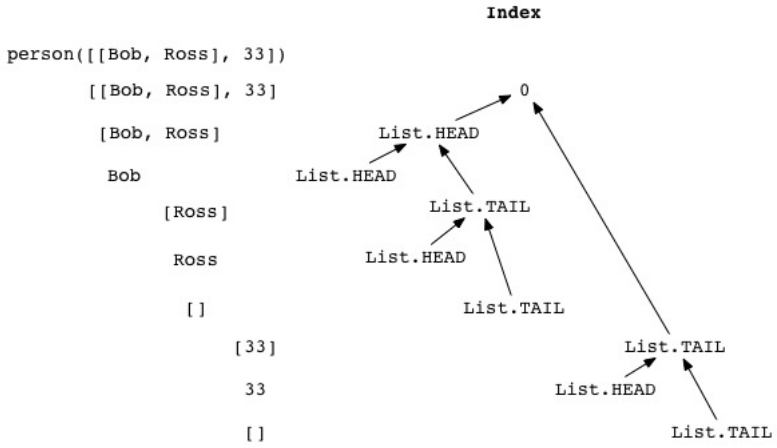


Figure 6.4: List Indexing

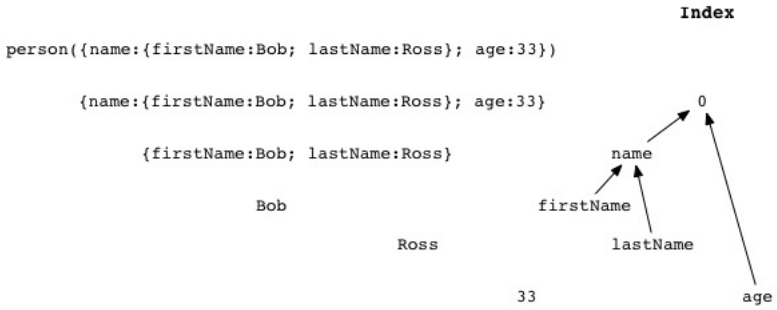


Figure 6.5: Record Indexing

listing 6.11

Listing 6.11: CRIME - Rule older than

```
olderThan(?name, Bob) :-
        person({name:?name; age >: ?age}),
        person({name:Bob; age : ?age}).
```

In this rule a join node will receive a variable `?age` from the left side of the join node; the variable will be put under the constraint greater than, while from the right side the variable will be put under the equality constraint. Thus, we have to determine how the comparison will be done. In this example the resulting join node will check the following condition `right.?age >: left.?age`. From this example we determine that at least in order to create a join node, one of the variables should be put under the equality constraint.

We subsequently explain how the Rete network is constructed by means of the following example:

*Bob attends a domino club every Wednesday. Since there is an upcoming domino championship, he is trying to find a good team-mate to play with. In order to do so, he has defined a new rule for saving all personal information available of different players that attend the club. He filters their information based on the score and the age. He is looking for persons that have at least the same score he has, and that are around 30 - 40 years old.*

The rule is depicted in figure 6.12.

Listing 6.12: Rule with Rete complete description

```
:SaveContact({contact:?person; group:DominoClub}) :-
        personalProfile(?me),
        ?me.score : ?score,
        location(?me, DominoClub),
        service(?person),
        ?person.score >: ?score,
        ?person.age >: 30,
        ?person.age <: 40.
        .
```

In figure 6.6 we present a small representation of the rule presented in our previous example. We can identify that we never specify the score, however we make sure that the combination is possible, since we have a way for binding the value of score (`?me.score:?score`). The variable `?me` should be a record because the expression `?me.score : ?score` indicates that it contains a pair which key is "score", and the same for the person variable. We identify that filtering a compound attribute means to filter each of the attributes that form it, as shown in the figure.

### 6.3.4 Pickers

`AttributePicker` class permits that production nodes get from tokens attributes. These attributes are parameters for the action. We extend the `AttributePicker` class for the new structures. Each picker of a compound attribute is formed by a vector of attribute pickers, that contains all the pickers for each attribute that conforms the compound attribute.

### 6.3.5 Persistent Facts

In chapter 4 we have highlighted the need to provide mechanisms to save context information in order to allow users to define rules later and not annoying them with the definitions of new rules each time an action takes place or whenever new context information is available. We have mention that CRIME gives a consistent view of the environment by means of asserting facts that show the availability of services, thus when the device disconnection takes place, the facts,


Figure 6.6: Rete with records and Relational operators

that were exchanged with it, are retracted. However there are context information that could be useful to keep. In order to do so, we have added the persistent construct that enables to add a fact but without any retraction when its prerequisites are not longer available. The persistent construct receives as arguments a set of facts with their corresponding arguments in each fact. The arguments of each fact can be variables that have to be bounded with the rule's prerequisites. Listing 6.13 shows an example of using the persistent construct. This example creates a persistent fact (`knownService`) with the same argument (`?service`) as the `serviceAvailable` fact and that has not been created yet.

Listing 6.13: Persistent construct

```
persistent(knownService(?service).) :-
        serviceAvailable(?service),
        not knownService(?service).
```

Using the persistent construct we allow to determine which information is relevant to keep and the way it will be saved. It is also possible to save context information and share it by means of the fact space model (e.g. `persistent(public`

-> knownService(?service).)). And after the creation of a new persistent fact, it is possible to define rules using it as any other fact.

The Rete network for this construct is the normal: where filter nodes are created for each attribute that are not variables and join nodes are created for variables presented in different patterns. The persistent expression is an extension of the Event class. This event conforms the action part of a rule. Instead of attributes it receives an ExpressionList of facts to create. The only difference is the iteration process in order to create pickers for each attribute of each new persistent fact to create.

### 6.3.6 History Context

It is possible to retrieve saved facts by means of the findall statement, where we filter based on a predefined fact kind. For instance we can create a persistent fact for each service that we encounter, as shown in listing 6.14. Then using findall we can get all the attributes that constituted the previous events by assuming that all facts we received have the same kind.

Listing 6.14: History by means of findAll

```
persistent(knownService(?service).) :-
        serviceAvailable(?service),
        not knownService(?service).
:DisplayHistory(?services):-
        findall( ?service,
                ( knownService(?service). ),
                ?services ).
```

However this form can be restricted since we have to save the facts using a single fact kind (knownService), or if we want to have different kinds, to get all we need different rules with findall statements. Also we need to know the arity of each fact that constitutes the history. We have added the history construct which enables to get all history context. We consider all persistent facts plus current context as history context. This construct has three parameters. The first one enables to filter context history based on the fact space, the second filters the fact kind, and the third which is used as an accumulator variable where the facts that conform the history context are returned as a list of facts. The Rete network for this construct is created by using two filter nodes just if they are needed. One filtering the fact space and another filtering the fact kind. Finally we have added an AccumulatorNode that accumulates all occurrences of certain facts (filter by fact space and fact kind). This node contains a cache which stores all the facts that contitute the context history, according to the filtering specified in the construct. Whenever the cache is updated, the previously sent token to the children must be retracted. For instance, consider that it is the first time Bob uses his Pda that is context-aware, and he wants to save all the printer services available as history. Listing 6.15 illustrates the defined rules. The first rule indicates that he wants to save all printer services that he finds. While the second enables him to get all the

printers that he has saved, as part of the history context. This last rule is triggered each time a new persistent fact with the `knownPrinter` kind is saved.

Listing 6.15: History construct

```
persistent(knownPrinter(?printer).) :-
        printer(?printer),
        not knownPrinter(?printer).
:DisplayPrintersSaved(?printers):-
        history(private, knownPrinter, ?).
```

### 6.3.7 Current Context

This construct enables to get the available context of an application. Building the Rete network for a current expression is very similar to that of the history expression. The construct has three parameters. The first one enables to filter current context based on the fact space, the second one filters the fact kind, and the third one which is used as an accumulator variable where the facts that form the current context are returned. We make use of the Rete network to avoid iterating and filtering over a memory in the root node. We make the assumption that all facts that are not persistent are considered as part of the current context. This assumption is what makes the difference in the Rete network, which needs an extra filter node that filters only facts that are not persistent.

## 6.4 Conclusions

In this chapter we have presented our extensions to the CRIME language. Theses extensions provide a way to describe the world, based on composing and structuring data. We have added two different data structures: lists and records. Using records we provide a free and sufficiently generic structured way for defining context information. We use the open term to indicate that it is flexible, extensible and adaptable to constantly changing parameters and the type of information we are describing with it. It has minimal barriers to adoption by the environment. We have also presented the most important aspects of the implementation of this extension. First the parsing of the structures and then the required extension of the Rete network to support them.

In the following chapter we explain the accessible methodology for end-users. CRIME, augmented with the extensions presented in this chapter is used to illustrate the methodology in the following chapter.

# 7 An Accessible Methodology

In this dissertation, we argue for an accessible methodology to involve the user in the development of context-aware applications by means of an open ontology to describe the world. In chapter 3 we have shown that context-aware systems focus on facilitating the task of developers during the implementation of context-aware applications. However, the context reasoning, that determines when an application should adapt, is hard coded in the application base code. This way, current context-aware applications try to anticipate all possible scenarios and propose fixed adaptation for them. These scenarios are designed by developers, based on their experience and intuition. However, developers may not know which response is more appropriate in a particular context. End users are the ones who really know how their application should behave. Moreover, different users interact differently with the environment. Additionally, user needs change and evolve over time: using fixed scenarios does not allow context-aware applications to follow this evolution. Also, users feel they lose control over their applications [BD03], since their application can behave in an unexpected way nor do they exactly know the reason for their applications' behavior. Thus, if context-aware applications fail anticipating the users' desired behavior, this in general leads to frustration and nonacceptance of context-aware applications.

In order to give the users control over their context-aware applications, approaches like the ones presented in chapter 4 have emerged. We have highlighted some properties that are common, and others that are desirable to have. We identify a need for an abstraction level that enables end-users to interact with their context-aware applications. Also, they have to be provided with a discovery mechanism to be aware of what kind of services are available. In order to have real scenarios where users define the adaptations they want; users must be provided with mechanisms to allow them to create their own scenarios with the corresponding behavior. Furthermore, users should not be disturbed during their activities in order to define new rules that determine the behavior of their applications. This implies the

need for the storage of history context, to enable users to define their rules at a later moment. In order to define the scenarios, users may interact with all the real and possible context information. Once defined the rules should be verifyable, so that users can check whether the application's behavior is as desired, and if needed redefine them. Some intelligence could be added, but it should be controllable, in order to avoid unexpected behavior. In this chapter we describe which are the principles of the methodology we propose and how to use CRIME augmented with the extensions described in the previous chapter, in order to configure, develop and design context-aware applications. We make use of the mobile phone environment, for illustrating the latter.

## 7.1 Developer concerns

There are many scenarios that developers cannot anticipate and some assumptions about the desired behavior will not match the users' expectations. The development of context-aware applications requires more than defining specific scenarios and associated reactions. The design decisions should include aspects that permit users to interact smoothly with their applications and hand the control to them.

### 7.1.1 Abstraction Level

In order not to confuse or annoy the user, context information must be abstracted and filtered. For instance, if we provide users with the current coordinates given by their GPS sensor, this information may be not understandable for them. This way we want to limit the information and separate low-level information from high-level information. Developers provide a library of mappings of low-level to high-level context. This higher-level abstraction shields users from lower level data, which concepts may not be clear, or comprised by overly detailed or irrelevant information. In the given example, users might treat locations by name rather than by the coordinates provided by their location sensor. In general, the level of abstraction depends also on the type of application at hand. Listing 7.1 shows the kind of rules that we refer, using the location rule.

Listing 7.1: Abstraction Level Rule

```
location({person:?person; at:?place}) :-
    coordinates({id:?id; x:?x, y:?y, name:?place}),
    person({id:?id; name:?person}).
```

In this example we use the records presented in the previous chapter in order to clarify the semantics of each attribute.

#### Past and current context

Users interact in different environments, where each of them contains relevant context information that is available just at the time the users are in the connection

range. We propose to provide some control by means of a discovery mechanism that enable users to know which is the current context; the current context is the set of context information currently relevant and available.

Expecting users to define their rules the moment that the context information is available, could be annoying for them. In order to enable users to define the scenarios that fit their needs the moment they want to do so, contexts should be stored. As such, when the context is stored in a context history, this context history can be used at a later time to define new scenarios for adaptations. And linking particular context with certain actions, will enable users to see what they have done before in similar situations or in which context he have repeat the same adaptation.

During our CRIME implementation we have enable to filter history and current context based on the fact space type, since we are considering that not only context information can be shared also history context. However different considerations must be taken into account. Given the limited resources of target systems, developers must define what and where context information should be stored by means of rules, or they can delegate part of this task to end-users and allow that they decide which kind of information they save. Also, some storage strategies must be provided: we refer as storage strategies keeping just a specific amount of context history, or overwrite the last context history information when the stack of context history is full.

Listing 7.2 shows a rule example that defines which kind of information we are storing in our example. This is done by the persistent construct that we introduce in chapter 6

Listing 7.2: Saving past context

```
persistent(serviceKnown(?service).):-
        service(?service),
        not serviceKnown(?service).
```

This way if a printer service is available, we saved it by means of a fact like the one presented in listing 7.3, but with the `serviceKnown` kind and we can have the information even if the service is not longer available.

Listing 7.3: Fact Example

```
service({printer:{
                id:P101;
                prints:color;
                ip:10.0.1.12
                }
        })
```

### 7.1.2 Adaptations

Adaptations are all the possible actions that an application can do in order to adapt to a certain context. Developers should define which actions should be available for users. In this step, developers have to define two sets of information: one dictating

the basic functionality the device should have, and the other set determining which actions can be used as adaptations by users. However, these two sets can intersect, which means that some actions can be shared in both sets.

Visible actions for users should be provided as black boxes, hiding technical details of how the adaptation is implemented. These actions can have a set of mandatory parameters. These parameters consist of context information that it is required in order to make the adaptation. A very simple example is when we want to ignore incoming calls when we are at the meeting room, the mandatory context in order to ignore a call is obviously a call. The fact representing that we are at the meeting room is just to fulfill user expectations about where to ignore incoming calls, but it is not required to ignore an incoming call.

### 7.1.3 Default Behavior

Developers can provide some predefined adaptations with certain scenarios. However, users must be able to activate, deactivate or modify them in order to fit their needs if they do not. For instance in our cellphone example, developers can provide a rule that changes to silent profile each time the user enters a public place. Though this seems a useful rule, developers cannot guarantee if it is a desireable rule to have for *all* users. In this step we can add rules that do not take into account human aspects, (e.g. the display configuration (portrait or landscape) example that we have already mention).

## 7.2 User Interaction

Context information is available for users in a structured way and with the sufficient level of abstraction that allows users to be involved as they can manually modify the behavior of their applications. In addition they are assisted with how to define the scenarios that dictate the adaptations.

In our example we have provided a discovery list that includes all entities in the history. We have given them derived from the record representation a tree representation as shown in figure 7.1.

This representation enables to get the necessary entities to define their scenarios. Also we allow to adjust the values to create new rules.

### 7.2.1 Query by Example

Since most people are not able to program, we propose an interaction technique which resembles QBE (query-by-example) [Zlo77]. QBE is a language, that by means of a graphical interface, enables the management of a relational data base. Users use skeleton tables to get the required information by filling in the table spaces with examples of the desired retrieval. These table spaces are filled with constants and variables. In our case, we work with examples of context entities


Figure 7.1: Discovery List

instead of example tables. We introduce this approach by means of different examples. Initially users are presented with an example context entity which exhibits a skeleton and values for each characteristic as shown in figure 7.2. We implement this display where the first column refers to the name or key of a record pair, the second column enables to define if the key is mandatory and from the third to the end we can add different conditions.

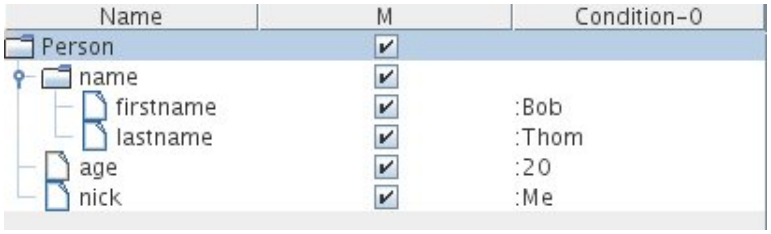


Figure 7.2: A context entity example

The figure 7.2 is a graphical representation for the facts listed in listing 7.4.

Listing 7.4: Fact examples

```
service({Person:
                {name:
                        {firstname:Bob;
                         lastname:Thom};
                age:20;
                nick:Me}
        }).
```

```
Person({name:
                {firstname:Bob;
                 lastname:Thom};
        age:20;
        nick:Me}).
```

Users may now change the row value to express the example context entity they want. It is possible to select or deselect key elements they do not consider relevant to filter. For instance if the user is just interested in persons who's first name is "Bob", then she might select the `firstname`. However since the `firstname` is part of the name and the name part of a person, then the parent keys (person and name) are also selected as shown in figure 7.3. Figure 7.3 also includes the age, which means that the age of a person is available, but that the value in this particular case is irrelevant.

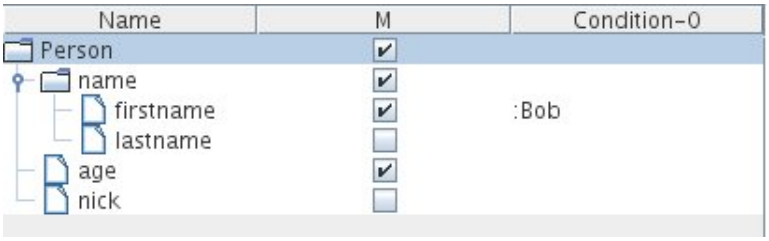


Figure 7.3: A context entity example with partial key selection

This way users interact with skeletons of real entities.

### Qualified filtering

Qualified filtering enables to specify conditions using operators such as: !:, >, :>, <:, <. They place restrictions on the attribute values. For instance, if we consider that we want to define a scenario where the battery level is between 0% and 20% we need a condition like the one depicted in figure 7.4.

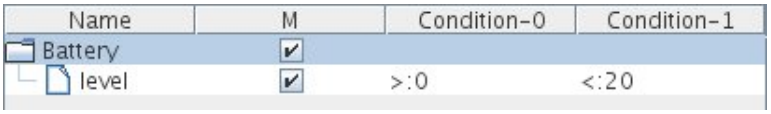


Figure 7.4: Qualified filtering

### Different entities

If a user needs to define a scenario that requires two or more entities, she may do so by adding the corresponding entities to the scenario and modify the values of

the generated skeletons. Consider for example that we define a scenario where the presence of Bob's boss (Ted) determines the action to ignore incoming calls from a particular number. This scenario involves two entities: *the presence of Bob's boss (Ted)* and *a call from the particular number*. Figure 7.5 illustrates this scenario.

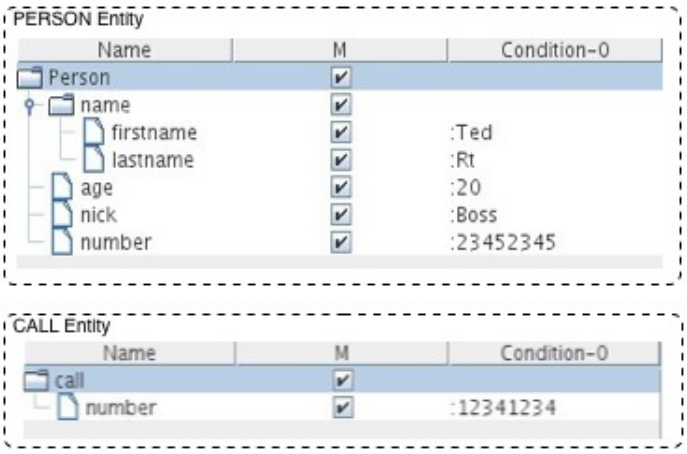


Figure 7.5: An entity for modelling Bob's boss and an entity for modelling an incoming call

This scenario shows that allowing users to define adaptations themselves, enables to define adaptations based on social aspects (e.g. a social relation as the `boss-employee` relationship). This is much harder to do when hard coding context adaptation logic.

### Links between different entities

We can establish links between two or more entities by means of variables. When two entities share the same variable for a particular attribute, then these attributes must have the same value for the entities to match. Consider a scenario where we want to add a contact to Bob's contact list but only when the contact is not already present in the contact list. Making sure that the person's name is not already present in the contact list can be done by linking the `contact` and `person` entities through the variable `?name`, as depicted in figure 7.6.

### Links in the same entity

The same variable can be used multiple times in the same entity, to denote that two keys should have the same value. For instance if we want to display all the contacts that have the same name as their nick name, we may use an instance like the one shown in figure 7.7.

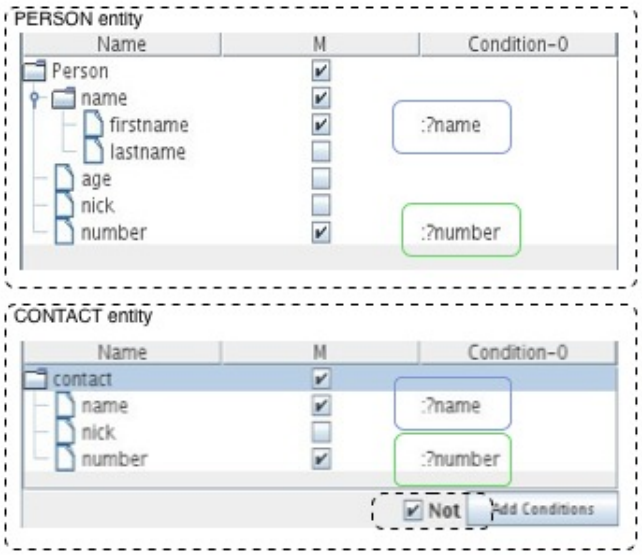


Figure 7.6: Link between different entities

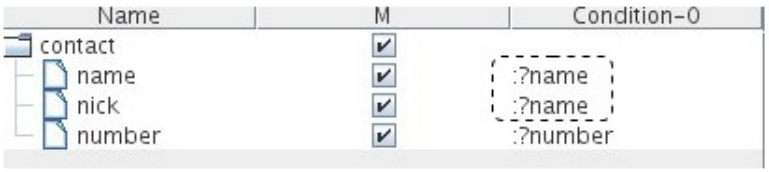


Figure 7.7: Links in the same entity

### Negation

Negation implies the absence of an entity. Variables in a negative entity must also appear in a positive entity. Consider the scenario where we want to add a contact to Bob's contact list if they not already exist in the contact list. Figure 7.6 already illustrate this scenario, by means of a not check box.

### Getting values for actions

In this last scenario, in order to add a new contact to the contact list, users need to specify from where to take the information needed to add a new contact. Adding a new contact requires the name of the person and the phone number. Users can pass variables like parameters to actions.

Using this approach, a user concentrates on which entities determine the behavior they want and the connections between them. Since developers already

filter and make the abstractions for the user, users are able to use this context information, and define rules that will dictate the desired behavior in the context they define. Notice that users define rules by using real entities. Allowing users to define their applications' behavior permits that these applications agree with the users' expectations.

## 7.3 Feedback Mechanism

The feedback mechanism supports understanding and gives users an idea of how and why their applications exhibit some behavior. Also, the feedback mechanism allows to maintain rules. It is quite challenging to define the right rules at once. Rewriting and redefining rules gives an initial solution to transform it in a better solution. This implies that some generalization or specialization of rules is needed.

In addition, a rule is considered relatively independ of other rules. However, different rules can be triggered by the same context. Hence some support to make the user aware of these cases and/or deal with them, needs to be incorporated.

### 7.3.1 Rule Generalization

Users may define a new rule each time they perform an action, using the current behavior, and without doing any modifications to it. Unfortunately, such rules may be to specific to be used in a broad set of situations. A rule can be generalized when a user applies an action already defined in another rule where the entities that constitute the scenario have a similar skeleton. If we have two similar situations where common context entities are taken into account and the action is the same, then users can create a generalized version of this rule.

In order to provide rule generalization and rule specialization, the history context is not sufficient. We need a link between the adaptations and a picture of the context at the moment the adaptation takes place. This allows to do a comparison of similar scenarios to define a good rule.

Consider the following scenario, Alice is Bob's client. Bob has created a rule that allows him to ignore incoming calls from the number 12341234 since it was the first scenario he had the first time when Alice was around. In listing 7.5 we show the rule.

Listing 7.5: IgnoreCall rule example

```
:IgnoreCall(12341234) :-
        call({number:12341234}),
        person({name:Alice;
                phone:23452345;
                address:{
                          street: "Place Carnoy";
                          number: 11;
                          zip: 1200
                        }
```

```
        }),
    profile({sound:Loud; tone:Tone1}),
    battery({level:70}).
```

However, during the day Bob ignores different calls. For example, if he receives a `call(number:56785678)` he ignored it because Alice was at same point around. He can identify that the rule he has just defined is to specialized to cover other possible scenarios. He can define a rule for each new scenario, however since the profile and the battery level constitutes part of the context, he will have to define a rule for each level of the battery if it is not contemplated. To generalize the rule, different actions can be done, first take out information that he does not consider as relevant for his scenario like the `profile` or the `battery`. However he can also make variables of this values for instance `profile(?profile)`. Then he has to notice that the actual phone number does not matter: he just wants to ignore all incoming calls when he is with Alice. Then he has to change `call(number:12341234)` to `call(number:?number)` and pass the `?number` variable as a parameter to the `IgnoreCall` action. More generalization is possible, if we have the contact information like: `contact(name:Alice; number:23452345; group:Client)` we can make a more generalized rule that applies for all clients, rewriting the rule as in listing 7.6.

Listing 7.6: Generalized rule IgnoreCall example

```
:IgnoreCall(?number) :-
        call(?call),
        ?call.number:?number,
        person(?person),
        ?person.name:?name,
        contact({name:?name;group:Client}).
```

### 7.3.2 Rule Specialization

If users find a rule that is successful in one situation and not in another, some specialization is needed to add conditions that make the rule more restrictive or not applicable in such a situation. Consider that Bob has generalized the rule as the one presented in listing 7.6. However he trusts Billy one particular client. So, Bob wants to be able to receive the calls even when his client Billy is around. The previous rule is general, but not good enough to catch this particular scenario. This way specialization is required in order to match this scenario. In order to do so, Bob needs to add the following condition: `?person.name!:Billy`. In this scenario we can see that allowing the user to define his own rules may create scenarios that could not be conceived in a hardcoded approach.

We have exemplified rule specialization and generalization by means of simple examples. However this task is not trivial for end-users. We propose a system based on inductive logic programming [Fla94] in order to suggest the user possible generalizations or specializations of rules. However our current extension does not

cover this. We provide means to get the current and history context as it was shown in chapter 6.

### 7.3.3 Conflicting Rules

A user's actual situation can trigger different rules at the same time: these may conflict with each other. Bob has defined two rules, one is dictating that whenever he is at the meeting room, his cellphone's profile switches to silent. The second rule switches the cellphone's profile to loud at lunch time. However, the behavior of the cellphone is not clear when we have tea at the meeting room during lunch time. Both rules are applicable in this scenario, but we cannot apply both. Users need mechanisms to determine which rule should apply in case of conflicting actions. Priority mechanism like in CADEL [NYS+05], presented in chapter 4, can be applied. However there are other conflict resolutions strategies that may be helpful like specificity taking the most specific rule.

## 7.4 Conclusions

In this chapter we propose a methodology to allow the development of context-aware applications that take into account end-user considerations. We have described the most relevant properties of such a methodology, which we have identified in the previous chapters. We propose the QBE approach in order to interact with users. This way, users are able to define their rules to suit their needs and determine how their applications should react in a given scenario. Rule generalization and specialization is an important task during the definition of new rules. Through these are not trivial tasks for end-users, it is possible to create a recommender systems that assist users in this process.

# 8 Conclusions

In this dissertation, we argue for an accessible methodology to involve the user in the development of context-aware applications with an open ontology to describe the world. During chapter 3, we have illustrated that when we look at current context-aware systems (Context Toolkit, JCAF, WildCAT, GAIA, LIME), we came to the constatation that end-users are not involved in the development of context-aware applications. Context-aware systems focus on facilitating the task of developers during the implementation of context-aware applications. These context-aware applications have fixed adaptation logic. This way, current context-aware applications try to anticipate all possible scenarios and propose an adaptation for them without taking into account users.

## 8.1 Contributions

While proving our claim, the following contributions were made:

- In chapter 4 we presented a survey of current approaches (CAMP, ACCORD, CADEL, CAPpella and iCAP) whose aim is to allow users to interact with their context-aware applications. We have evaluated them by means of the four principles (fluidity, intuitiveness, robustness and calmness) that dictate organic user interfaces. This survey constitutes part of the case of the considerations we take in our methodology.

- The first step towards user interaction is to have an expressive mechanism to provide contextual information. Unfortunately, the use of simple predicates like the ones used in GAIA or CRIME, have some restrictions: the semantic of each attribute it is based on an ontology that users may not know, positional dependencies, the arity must be fulfil and evolution constraints. In

order to diminish these restrictions, we made an extension to CRIME, as explained in chapter 6. This extension is based on structuring and composing contextual information by means of lists and records. These mechanisms provide a better understanding of contextual information inside facts, not only for users, but also for developers.

- We proposed an accessible methodology that involves the user in the development of context-aware applications. This methodology describes considerations for users during the development of context-aware applications. Developers have to reach beyond the particular implementation itself, and they have to design the means for user interaction. We can summarize this methodology in the following points:
  - Users must be provided with knowledge that allows them to have an understanding of what their application knows about the context. In order to provide this knowledge, an abstraction level is required, which is provided by a library of mappings of low-level to high-level information. Also means to save relevant context information that will enable users to used them latter.
  - Developers must provide well defined actions that users can take to define their own scenarios and associated adaptations.
  - A mechanism for rule definition where they associate context information, that represents their scenario, to an action. This way defining their own adaptations in the form of rules. We have proposed an approach similar to query by example that enables them to define and establish relationships between the entities involved in their scenarios.
  - Applications have to provide feedback. We have identified three problems that arise in rule definition: rules can be to general or to specific or/and they can conflict with other rules. Part of this feedback will enable to manage these three problems.

## 8.2 Future Work

The following points correspond to possible continuation of the work presented in this dissertation:

### Management of context history

We have added the persistent construct to the CRIME language to define which context information developers or users want to store as context history. Deleting this history implies the retraction of these persistent facts. Our current approach creates a persistent fact each time the preconditions of the rule are met. However, some other mechanisms can be provided in order to store significant information. For example: save just the first occurrence of an entity represented by a

fact, save the last one, save a particular amount of data (e.g. the first four), save the more specific entity (e.g. if we have two records, the one which has more pairs). This last case the action part of the rule may look like `persistentSpecific( knownPrinter( ?printer ).)`. Where if we receive the fact `printer( {id:100; ip:"10.0.1.10", prints:color; resolution:800;- paper:available; quality:photo} )`, first this rule will save this as a `knownPrinter` fact and when a new fact like `printer( {id:300; ip:- "10.0.1.7"; prints:black} )` is asserted it will not be saved since it is less specific than the first one.

### Support on the rule definition

Provide smarter mechanisms to maintain user rules. A recommender systems is suggested to support users to deal with the complexity involved in the rule generalization and rule specialization. A technique that can be use for this is inductive logic programming addressed in [Fla94].

Also different conflict resolution mechanisms exist (e.g. priorities, based on past activations , FIFO, random, etc.). However more investigation is needed, in terms of which or whose strategies are more suited for end-users, how they can specify them. In CRIME this will imply to add new constructs to specify the conflict resolution strategy. Like the priority construct of GAIA that we illustrate in chapter 3.

### User interaction

We have shown that organic user interfaces seem to fulfil the level of interaction required for context-aware applications. However, none of the current approaches presented in chapter 4 fulfils *all* of the principles (fluidity, intuitiveness, robustness and calmness) required for organic user interfaces. We have implemented a small interface for rule definition using our approach, resembling query-by-example however work has to be done in order to provide an interface that fulfils all the principles. Afterwards we would need a study with end-users to prove the effectiveness our approach.

### Complex cases

We have to perform more complex case-studies applying our methodology, than the examples investigated so far. This will enable to evaluate in different settings and larger ones the feasibility and effectiveness of our approach.

# Bibliography


[AAH+97] Gregory D. Abowd, Christopher G. Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: a mobile context-aware tour guide. *Wireless Networks*, 3(5):421–433, 1997.

[ADB+99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.

[BD03] Louise Barkhuus and Anind K. Dey. Is context-aware computing taking control away from the user? three levels of interactivity examined. In Dey et al. [DSM03], pages 150–156.

[Che03] Harry Chen. *An Intelligent Broker Architecture for Context-Aware Systems*. Phd proposal, University of Maryland, 2003.

[Cro06] D. Crockford. The application/json media type for javascript object notation (json). *Network Working Group, Rfc 4627*, July 2006.

[DBS+01] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J-C. Burgelman. Istag: Scenarios for ambient intelligence in 2010. Technical report, IST Advisory Group (ISTAG), 2001.

[DHB+04] Anind K. Dey, Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. A cappella: programming by demonstration of context-aware applications. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 33–40, New York, NY, USA, 2004. ACM Press.

[DL05] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[DSA01] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human Computer Interaction*, 16(2-4):97–166, 2001.

[DSM03] Anind K. Dey, Albrecht Schmidt, and Joseph F. McCarthy, editors. *UbiComp 2003: Ubiquitous Computing, 5th International Conference, Seattle, WA, USA, October 12-15, 2003, Proceedings*, volume 2864 of *Lecture Notes in Computer Science*. Springer, 2003.

[DSSK06] Anind K. Dey, Timothy Sohn, Sara Streng, and Justin Kodama. icap: Interactive prototyping of context-aware applications. In Kenneth P. Fishkin, Bernt Schiele, Paddy Nixon, and Aaron J. Quigley, editors, *Pervasive*, volume 3968 of *Lecture Notes in Computer Science*, pages 254–271. Springer, 2006.

[E.05] Bardram Jakob E. The java context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications. In Hans-Werner Gellersen, Roy Want, and Albrecht Schmidt, editors, *Pervasive*, volume 3468 of *Lecture Notes in Computer Science*, pages 98–115. Springer, 2005.

[Fla94] Peter Flach. *Simply logical: intelligent reasoning by example*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[For82] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[HCH+03] Jan Humble, Andy Crabtree, Terry Hemmings, Karl-Petter Åkesson, Boriana Koleva, Tom Rodden, and Pär Hansson. Playing with the bits: User-configuration of ubiquitous domestic environments. In Dey et al. [DSM03], pages 256–263.

[MPR01] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, volume 00, page 0524, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[MSP+07] Stijn Mostinckx, Christophe Scholliers, Eline Philips, Charlotte Herzeel, and Wolfgang De Meuter. Fact spaces: Coordination in the face of disconnection. In Amy L. Murphy and Jan Vitek, editors, *COORDINATION*, volume 4467 of *Lecture Notes in Computer Science*, pages 268–285. Springer, 2007.

[NYS+05] Kouji Nishigaki, Keiichi Yasumoto, Naoki Shibata, Minoru Ito, and Teruo Higashino. Framework and rule-based language for facilitating context-aware computing using information appliances. In *ICDCSW '05: Proceedings of the First International Workshop on Services and Infrastructure for the Ubiquitous and Mobile Internet (SIUMI) (ICDCSW'05)*, pages 345–351, Washington, DC, USA, 2005. IEEE Computer Society.

[RC03] Anand Ranganathan and Roy H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal Ubiquitous Comput.*, 7(6):353–364, 2003.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM Press.

[SP07] Christophe Scholliers and Eline Philips. Coordination in volatile networks. Master's thesis, Vrije Universiteit Brussels, June 2007.

[SS86] Leon Sterling and Ehud Shapiro. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.

[TAB01] Khai N. Truong, Gregory D. Abowd, and Jason A. Brotherton. Who, what, when, where, how: Design issues of capture & access applications. In Gregory D. Abowd, Barry Brumitt, and Steven A. Shafer, editors, *Ubicomp*, volume 2201 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2001.

[THA04] Khai N. Truong, Elaine M. Huang, and Gregory D. Abowd. Camp: A magnetic poetry interface for end-user programming of capture applications for the home. In *Ubicomp*, pages 143–160, 2004.

[Wei95] Mark Weiser. The computer for the 21st century. In *Human-computer interaction: toward the year 2000*, pages 933–940, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[WHFG92] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The active badge location system. *ACM Trans. Inf. Syst.*, 10(1):91–102, 1992.

[Zlo77] Moshé M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.