Towards a Rule-based Approach for Context-Aware Applications

Laura Maria Daniele

Thesis for a Master of Science degree in Electronic Engineering from the University of Cagliari, Italy

Graduation Committee:

MSc P. Dockhorn Costa

Dr. L. Ferreira Pires

Prof. D.D. Giusto

Enschede, The Netherlands

May, 2006

Abstract

Context-awareness is a computing paradigm in which applications can sense and explore users' context in order to provide them with proper and useful services. These services are dynamic and able to satisfy the users' current needs based on context changes. Context-awareness implies intelligence that enables an application to discover, reason, and predict a situation, and adapt to it in a dynamically changing environment.

Context-aware applications can determine their behavior by sensing and exploring the user's content without explicit user intervention. They can intelligently react upon changes in the user's context performing actions relevant to the user, the application itself, and the interaction between user and application.

The Event-Control-Action (ECA) pattern is an architectural pattern that can be applied beneficially in the development of context-aware applications, since it presents solutions for recurring problems associated with managing context information and proactively reacting upon context changes.

The ECA pattern divides the tasks of gathering and processing context information (*Event* module), from tasks of triggering actions in response to context changes (*Action* module). These separate tasks are realized under the control of an application behavior description (*Control* module), in which reactive context-aware application behaviors are described in terms of ECA rules, which have the form *if*<*condition*> *then* <*action*>. The condition part of an ECA rule specifies the situation under which the actions are enabled, and it is composed by logical combinations of *events*. Events model some happening of interest in our application or its environment. The action part of the rule is composed by one or more actions that are triggered whenever the condition part is satisfied.

This thesis proposes support for the design and implementation of a controlling service for context-aware applications (the controlling service is offered by the Control module of the ECA pattern) by using Jess, a tool for developing rule-based systems. Rule-based systems emulate human expertise in well-defined problem domains by using a knowledge base expressed in terms of rules. Jess provides the controlling service with a rule engine component, which executes ECA rules and allows context-aware applications to react when something of interest occurs in the context.

A distinctive characteristic of our rule engine component is that it can process ECA rules expressed in a Domain-specific Language, coined ECA-DL, specially developed for context-aware applications. In this way we allow ECA rules written in ECA-DL to be executed in the robust and powerful environment of Jess.

Table of contents

Preface

This thesis arises from the collaboration between the University of Cagliari and the University of Twente by means of the Socrates/Erasmus program. In particular, this thesis describes the result of a Master of Science assignment at the Architecture and Services of Network Applications (ASNA) group at the University of Twente. This assignment was carried out from October 2005 to May 2006 as part of the Freeband AWARENESS project (http://awareness.freeband.nl). Freeband is sponsored by the Dutch government under contract BSIK 03025.

I would like to express my gratitude to all those who gave me the possibility to complete this thesis.

I would like to thank my Professor Daniele Giusto for suggesting the University of Twente as a suitable and qualified place where to do my Master project.

I would like to give my thanks to my Professor Luís Ferreira Pires for granting me the opportunity to work at the ASNA group of the University of Twente. I also wish to thank him for his persevering help, stimulating suggestions and encouragement that helped me throughout the development of this work.

In particular, I am very grateful to my supervisor Patrícia Dockhorn Costa who was a precious and irreplaceable guide with her experience and great preparation, patience and persevering attention, stimulating collaboration and, above all, her friendship.

I would also like to thank my colleagues at the ASNA group for the pleasant working environment and all the friends I made, especially those from Witbreuksweg 399, for being my family in those months far away from home, and for making my stay in the Netherlands the best experience of my life.

I am also very grateful to my parents who have always supported me in my university career with their patient love, and who backed me up in my Erasmus experience by being close to me with their emotional support.

Finally, I would like to dedicate this thesis to my grandmother “nonna Fabia” who is the brightest star in the sky and always reminds me, even in the most difficult moments of my life, that:

“Beyond the clouds there is always the sun…”

Laura M. Daniele

1 Introduction

This chapter presents the motivation, the objectives, and the structure of this thesis. It identifies the relevance of context-aware applications and draws special attention to an architectural pattern, the Event-Control-Action (ECA) pattern. This pattern provides a high level structure for context-aware applications that proactively react upon context changes.

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

1.1 Motivation

We are witnessing nowadays an unprecedented use of Internet, communication and computing technologies everywhere: in commerce, business, government, health, defense and educational applications. Advances in software technology, ubiquitous devices and the increasing volume of digital knowledge, offer the opportunity for more sophisticated and user-friendly digital services [7].

Computation is now packaged in a variety of devices. Smaller and lighter laptops, as powerful as conventional personal computers, free us from the confines of the single desk. Specialized devices such as handheld personal organizers are portable enough to be with us all the time. Wireless technology allows devices to be fully interconnected with the electronic world. Cameras and VCRs are being supplanted by digital equivalents. Mobile phones are really networked computers.

On a different scale, computation is also moving beyond personal devices. Interconnected computing devices, large and small, along with various sensing technologies, from simple motion sensor to electronic tags or to video cameras, are being used to make physical rooms and buildings “intelligent”. Interaction with computation can soon be an “environmental” and communal experience rather than just a virtual and private one. Through these developments, computation is invading the fabric of our personal and social activities and environments [8].

We are being carried in this direction by several related strands of research, beginning from ubiquitous computing (now often called pervasive computing), in combination with mobile and distributed computing, augmented reality, wearable computers and human-computer interaction. For better understanding, we briefly characterize these fields of research [2]:

- *Ubiquitous computing* integrates computation into the environment, rather than having computers as distinct objects. Another term for ubiquitous computing is *pervasive computing*. One of the goals of ubiquitous computing is to enable devices to sense changes in their environment and to automatically adapt and act based on these changes, also considering user needs and preferences.

- *Distributed computing* studies the coordinated use of physically distributed computers. The main goal of a distributed computing system is to connect users and resources in a transparent, open, and scalable way.

- *Mobile Computing* is a generic term to denote the application of small, portable, and wireless computing and communication devices.

- *Augmented reality* (AR) is a field of computer research which deals with the combination of real world and computer generated data.

- *Wearable computing* is an active topic of research, with areas of study including user interface design, augmented reality, pattern recognition, use of wearables for specific applications or disabilities, electronic textiles and fashion design. Wearable computers are small portable computers designed to be worn on the body during use.

- *Human-computer interaction (*HCI) is the study of interaction between people (users) and computers.

These technologies have in common that they move the site and style of interaction beyond the desktop (in both senses: virtual desktop, i.e., the graphical user interface with its desktop metaphor, and physical desktop where computing devices have been confined) and into the larger real world where we live and act. This presents many challenges since the desktop is a well-understood, well-controlled environment, and the real world, instead, is complex and dynamic. The main challenge is to make computation useful in the myriad of situations that can be encountered in the real world, the ever-changing *context* of use.

1.1.1 Context-aware applications

In this thesis we are particularly interested to context-aware applications, which is a term commonly understood by those working in ubiquitous/pervasive computing. In this area of research, context is felt as key in the efforts to disperse and enmesh computation into our lives. One goal of context-aware applications is to acquire and utilize information about the context of a device to provide services that are appropriate to the particular people, place, time and events [8].

Context refers to the physical and social situation in which computational devices are embedded. In another way, context can be defined as:

> *" The interrelated circumstances in which something exist or occurs."*

Therefore, context always holds a subject, or entity, which can be a user, a group of users, an object, or a service. In this thesis, we have embraced the following informal definition of context, which has been used as a reference in the literature of context-aware computing domain [9]:

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

Context-awareness is a computing paradigm in which applications can sense and explore users' context in order to provide them with proper and useful services. These services are dynamic and able to satisfy the users' current needs based on context changes.

Context-awareness implies intelligence that enables an application to discover, reason, and predict a situation, and adapt to it in a dynamically changing environment. Applications operating in distributed environments would also need to become mobile, in particular when servicing people on the move. In order to produces real awareness in ubiquitous/pervasive computing, programs with embedded intelligent must become mobile and retrieve context-related information in different locations. Thus, mobility aids in the intelligent acquisition of context [7].

Context-aware applications are applications that can autonomously determine their behavior by exploiting the user's context, i.e., these applications do not require explicit user intervention.

1.1.2 Context-aware services

Services offered by context-aware applications are context-aware services.

In systems which are highly distributed and may have huge amount of users, it is advisable to use a shared infrastructure to support context-aware applications. Actually, it is not feasible for each individual application to capture and process context information just for its own use (it increases costs and complexity of the system) [10]. Therefore, to support context-aware mobile applications we can use a context-aware infrastructure based on Service-Oriented Architecture (SOA), which is an architectural style that can be applied in the design of distributed applications.

Service-oriented architecture is characterized by the concepts of service user and service provider. The service concept concentrates on the behavior that can be experienced by the environment (users) of a system. A service specification consists of the interactions between the system providing the service and the users of this service, and the relationships between these interactions. This separates the service (the supported behavior) from the entity providing the service (the system) [11].

A context-aware infrastructure can be structured in a services layer and a networking layer, allowing networking issues to be shielded from service provisioning, and facilitating the usage of services by applications. In this way, specific applications of a particular domain, for example, in the health domain, may not need to interact directly with the networking layer but only with the components of the services layer. Each of these components offers its capabilities

as a service to other components, and a component can make use of the capabilities of other components by invoking their services.

In the development of context-aware applications we have to face some challenges such as context discovery, sensing, extraction, manipulation and interpretation. Actually, we need to create a correspondence between objects in the real world and objects in the applications, but circumstances sensed by the environment in real world can not be directly used by the applications. Therefore, it is necessary to model these *circumstances*, which form the context, in *context information*, characterized by well-specific values, which we call conditions.

Figure 1 shows the context of a person (application user) in real world and context-aware applications that can only refer to this context through context information [10].

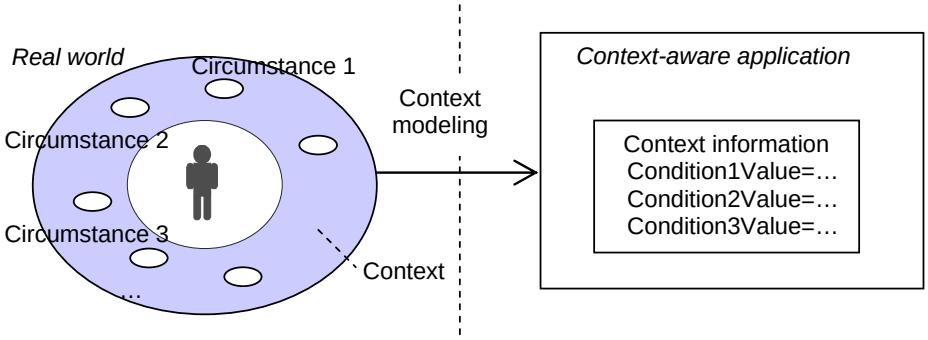

Figure 1 - Context in real world versus context information in context-aware applications

We can define many different kinds of context circumstances. For example, the geographical location in which the user can be found; or environmental circumstances of the physical environment of the user, such as temperature, humidity, light, etc.; or user's vital signs like the heart beat or the blood pressure.

1.1.3 ECA pattern

If some specific circumstances change in the user's context, the applications should be able to consequently adjust their behavior. For this purpose we can use the Event-Control-Action (ECA) architectural pattern.

The ECA pattern is based on condition rules, which we call ECA rules. These rules have the form *if <condition> then <action>*. The condition part is represented by logical combinations of events, and it specifies the situation under which the application has to perform proper actions, which can be a web service call, an SMS message delivery, or a complex composition of service invocations.

The ECA pattern consists of three components: the *Context Processor*, the *Controller*, and the *Action Performer*.

- The Context Processor observes and notifies events that occur in the context;

- The Controller component is provided with applications behavior descriptions (condition rules), and it observes events, monitors conditions rules and triggers actions when a condition is satisfied; and

➢ The Action Performer implements the actions and customizes services delivery as needed by the user and his context.

1.2 Objectives

The main goal of this thesis is to provide support for the design and implementation of a controlling service in the scope of the Freeband AWARENESS project [12]. The controlling service is offered by the ECA Controller component.

The Freeband AWARENESS project is concerned with the research and the design of a service and network infrastructure for context-aware mobile applications. The services and network infrastructures are validated through prototyping with ambient intelligent medical applications [11].

Figure 2 depicts a layered view on the AWARENESS architecture. The AWARENESS infrastructure consists of *Generic context-aware services* and *Context-aware session & mobility support services*. These context-aware services are built on top of the *Core middleware* and rely on *Network system resources*. The AWARENESS infrastructure supports *M-health services*, which form the basis for *Context-aware m-health applications,* such as, e.g., epilepsy detection. M-health applications are specific for a particular domain (the health domain), but, in a similar way, applications in other domains such as office and entertainment could be supported by the AWARENESS infrastructure.

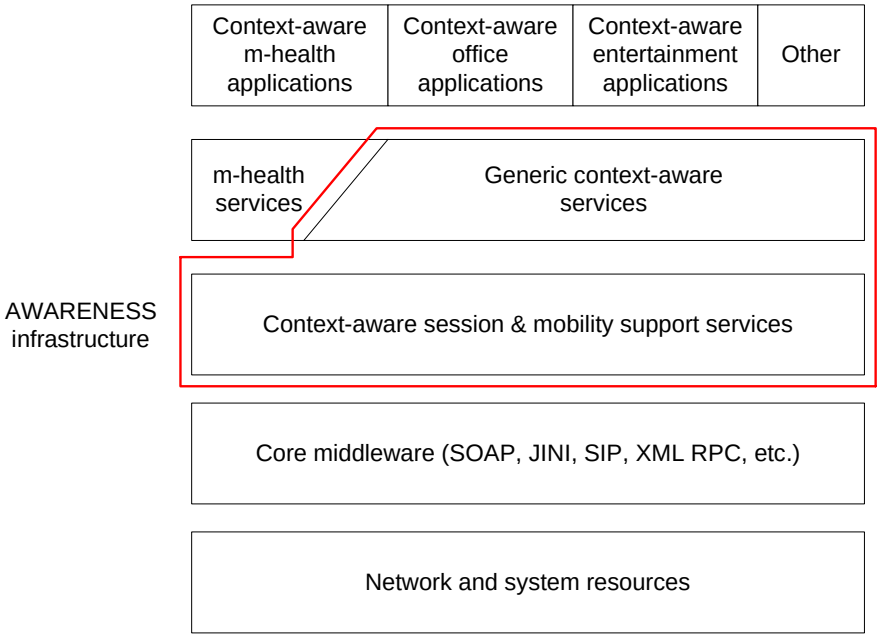

Figure 2 - Layered view on the AWARENESS architecture

In the scope of the AWARENESS project, our work aims at supporting the design of a controlling service that allows clients of this service to activate Event-Control-Action (ECA) rules and query for specific instances of context information. ECA rules define the situations under which actions should be triggered. Events define some happening of interest, conditions describe a combination of events, and actions specify the responses required when the conditions are met. Potential clients of the controlling service are applications that

would like to activate ECA rules within the AWARENESS infrastructure. Applications use this service to get event notifications back from the infrastructure.

1.3 Approach

Our efforts towards the design and the implementation of the AWARENESS controlling service include:

- The analysis of the Event-Control-Action (ECA) pattern, which can be applied beneficially in the development of context-aware services platforms;

- The study and presentation of the specification language (ECA-DL language), which is used to define ECA rules. These rules are used as input by the Controller component to configure the AWARENESS infrastructure accordingly. ECA rules allow the specification of context information events and respective actions that should be triggered in response to these events;

- The extensive study of rule-based systems and the definition of criteria in order to choose an available tool for developing rule-based systems. This tool should provide the controlling service with a rule engine component, which executes ECA rules and allows the AWARENESS infrastructure to react when something of interest occurs in the context;

- The extensive study of the chosen tool;

- The mapping of the ECA-DL language to the specific language adopted by the chosen rule engine in order to allow the engine to execute ECA rules expressed in the ECA-DL language.

1.4 Structure

The structure of this thesis reflects the issues that have been dealt with throughout the research process. This thesis is further structured as follows:

- Chapter 2 reports on the Event-Control-Action (ECA) pattern and its importance for context-aware applications. This chapter identifies the essential requirements to be satisfied by the ECA pattern;

- Chapter 3 presents the ECA-DL language, which is detailed in terms of its clauses, syntax and semantics;

- Chapter 4 presents an overview of expert systems based on rule engines, and selects some criteria for comparing different existing tools for developing rule-based systems;

- Chapter 5 presents some well-known rule engines, discusses them on the light of the criteria previously defined, and justifies the choice of one of these rule engines, namely Jess, for our work;

- Chapter 6 places special attention to Jess, the chosen rule engine. The chapter describes how Jess works for ECA rules and presents some usage examples;

- Chapter 7 deals with the mapping of the ECA-DL language, considered in Chapter 3, onto Jess. Particularly, this chapter presents examples of how ECA-DL rules can be mapped onto Jess rules, and proposes guidelines for the design of a generic mapping from ECA-DL onto the Jess language;

- Finally, Chapter 8 presents our conclusions and final remarks and identifies topics for future work.

2 Event-Control-Action (ECA) Pattern

This chapter presents the Event-Control-Action (ECA) architectural pattern. This pattern provides a high level structure that helps in the design of context-aware applications. The ECA pattern reflects the reactive nature of context-aware applications, whose behaviors can be expressed as ECA rules.

The chapter is structured as follows: section 2.1 discusses the relevance of architectural patterns and, particularly, of software architectural patterns. Section 2.2 presents the Event-Control-Action architectural pattern, by discussing its structure and dynamics. Sections 2.3 to 2.5 analyze each module of the ECA pattern. Particularly, section 2.3 analyzes the Event module, section 2.4 the Control module, and section 2.5 the Action module. Finally, section 2.6 presents conclusions about the benefits of using the ECA pattern.

2.1 Architectural Patterns

Architectural patterns have been proposed in many domains in order to capture recurring design problems that arise in specific design situations. These patterns document existing, well-proven design experience, allowing reuse of knowledge gained by experienced practitioners [24].

A software architectural pattern describes a particular recurring design problem and presents a generic scheme for its solutions. The solution scheme contains components, their responsibilities and their relationship. Patterns for software architecture also exhibit other desirable properties [24]:

- They provide a common vocabulary and understanding for design principles;
- They are a means for documenting software architectures;
- They support the construction of software with defined properties;
- They support building complex and heterogeneous software architectures;
- They help managing software complexity.

The Event-Control-Action (ECA) pattern is an architectural pattern that can be applied beneficially in the development of context-aware applications, since it presents solutions for recurring problems associated with managing context information and proactively reacting upon context changes [25].

2.2 The ECA Architectural Pattern

Context-aware applications can determine their behavior by sensing and exploring the user's content without explicit user intervention. They can intelligently react upon changes in the user's context performing actions relevant to the user, the application itself, and the interaction between user and application. These reactive behaviors of context-aware applications can be represented by using ECA rules that follow the Event-Control-Action (ECA) architectural pattern.

The ECA pattern divides the tasks of gathering and processing context information (*Event* module), from tasks of triggering actions in response to context changes (*Action* module). These separate tasks are realized under the control of an application behavior description (*Control* module), in which reactive context-aware application behaviors are described in terms of ECA rules, also called *condition rules*, which have the form *if*<*condition*> *then* <*action*>. The condition part of an ECA rule specifies the situation under which the actions are enabled, and it is composed by logical combinations of *events*. Events model some happening of interest in our application or its environment. The action part of the rule is composed by one or more actions that are triggered whenever the condition part is satisfied [25].

2.2.1 Structure

The ECA architectural pattern has been devised in order to decouple context management issues, such as sensing and processing context, from reaction concerns regarding reacting upon context changes, under the control of an application model. An application model defines the behavior of the application, which is described, in our case, by ECA rules (condition rules). Figure 3 shows the structure of the ECA pattern, its components and the relationships between them.

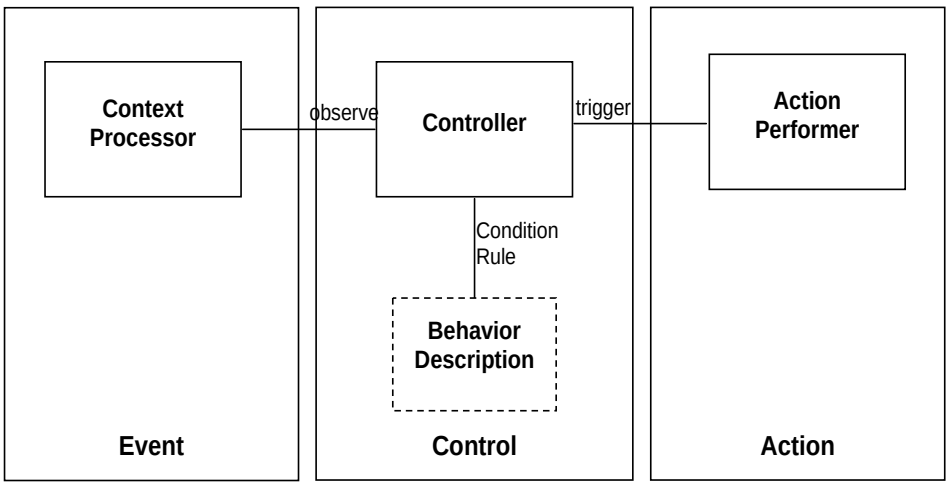

Figure 3 – Event-Control-Action pattern in context-aware applications

Context concerns are placed on the left side of Figure 3, which depicts the *Context Processor* component. This component depends on the definition and modeling of context information. The *Controller* component, positioned in the central part of the figure, is provided with application behavior descriptions, represented by the *Behavior Description* component. On the right side of the figure, the action concerns are addressed. The *Action Performer* component triggers actions, which can be a service invocation on (external or internal) service providers or a network [25]. Particularly:

- Events are modeled and observed by one or more Context Processor components;

- The Controller component, empowered with condition (ECA) rules describing application behaviors, observes the events;

- In case the condition of the ECA rule turns true, the Action Performer component triggers the actions specified in the rule.

2.2.2 Dynamics

We consider here a context-aware scenario in order to analyze the dynamic (behavioral) aspects of the ECA pattern.

Suppose that Patricia (supervisor) would like to be notified when Laura (student) is working. Since we cannot directly sense that Laura is working, we assume that she is working when she is in the laboratory without friends and her computer is on. According to the division in Event-Control-Action of the ECA pattern, we can consider the situation "Laura enters in the laboratory" as the event that triggers the evaluation of the following ECA rule:

If <Laura enters in Lab AND Laura is not with friends AND Laura's computer is on> *then* <Send notification (Patricia), "Laura is working">

The additional conditions "Laura is not with friends" and "Laura's computer is on" are needed to be sure that Laura is working and not, for example, speaking with friends. These conditions represent the situation under which the action of the rule is enabled in occurrence of the specified event "Laura enters in Lab". In case both the conditions returns true, the action "Send notification (Patricia)" specified in the *then* part of the rule is executed. Figure 4 depicts the flow of information between components in the ECA pattern for the scenario considered here.

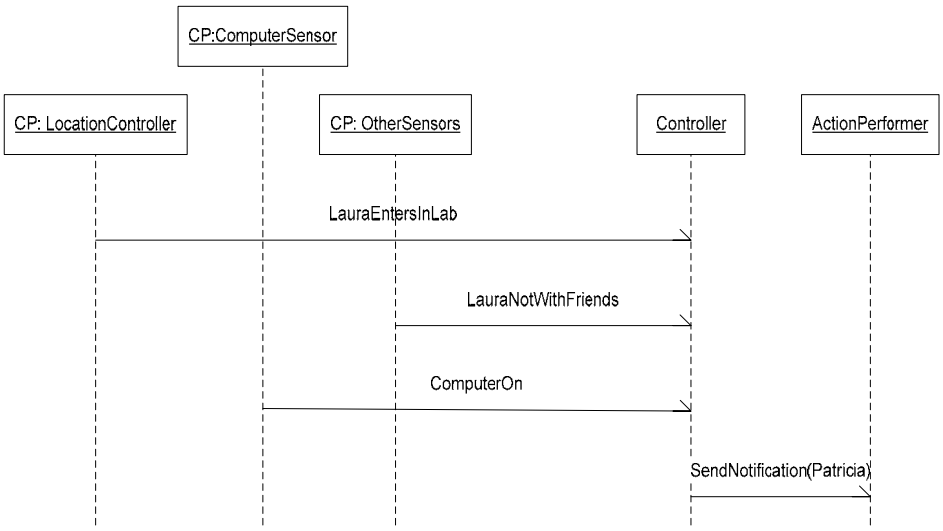

Figure 4 - Dynamics of the ECA pattern

The Controller observes the occurrence of event *LauraEntersInLab*. This event is captured by the component Location Controller, which is an instance of Context Processor. By exploiting sensors in the Lab, the Location Controller is able to sense when Laura enters in the lab. When this occurs, event *LauraEntersInLab* is generated.

Upon the occurrence of event *LauraEntersInLab*, the Controller evaluates the conditions *LauraNotWithFriends* and *ComputerOn*. This evaluation requires another cycle of context information gathering from other dedicated instances of Context Processor.

Finally, if both the conditions *LauraNotWithFriends* and *ComputerOn* return true, the Controller triggers the action *SendNotification* specified in the *then* part of the ECA rule and this action is executed by the Action Performer.

2.3 Event Module

The *Event* module of the ECA pattern is responsible for processing context information. Processing context information is challenging. Deducing rich information (e.g., Laura enters in lab) from basic sensor samples (e.g., a sensor that detects whether a person enters in the laboratory, and a recognition card provided with a microchip that Laura carries with her) may require complex computation. There may be several information processing phases needed before yielding meaningful context information [25]. Context information processing activities include [27]:

- Sensing: gathering context information from sensor devices. For example, gathering location information (latitude and longitude) from a GPS device;

- Aggregating (or fusion): observing, collecting and composing context information from various context information processing units. For example, collecting location information from various GPS devices;

➢ Inferring: interpretation of context information in order to derive another type of context information. Inference may be performed based on, for example, logic rules, knowledge bases, and model-based techniques. Inference occurs, for instance, when deriving proximity information from information on multiple locations;

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

Consider the Context Processor component of the *Event* module depicted on the left side of Figure 3. We can recursively apply context information processing operations in a hierarchy of Context Processors. In this chain of context information processing, the outcome of a context processing unit becomes input for a higher level unit in the hierarchy until a top-level unit is reached. For this aim, we define two types of Context Processor components, namely *Context Sources* and *Context Managers*.

Context Source components encapsulate single domain sensors, such as a sensor that detects when a person enters in a room. Context manager components cover multiple domain context sources, such as the integration of the detection that a person enters in a room and the recognition of this person. As shown in Figure 5, we can represent hierarchical chains of Context Sources and Managers as a *directed acyclic graph*, in which the initial vertexes (nodes) of the graph are always Context Source components and end vertexes may be either Context Sources or Context Managers. The directed edges of the graph represent the context information flow between these components [25].

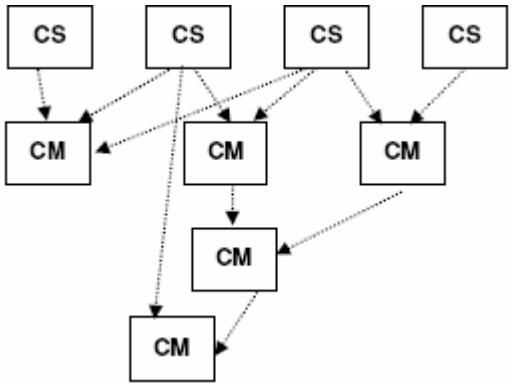

Figure 5 – Context Sources and Managers hierarchy in the Event module of the ECA pattern

Within a single context information processing unit (Context Source or Manager), we can verify recursive applications of the ECA pattern. Consider the same ECA rule of section 2.2.2 which is manipulated in Figure 6 by Controller C:

If <Laura enters in Lab AND Laura is not with friends AND Laura's computer is on> *then* <Send notification (Patricia), "Laura is working">

In this scenario, we assume that the laboratory is provided with a sensor for detecting when a person enters the laboratory, and that each person employed in the building (e.g., professors, PhD students, MSc students, etc.) is equipped with a

recognition card provided with a microchip that contains personal information. Moreover, we assume that there is a sensor in each computer able to detect whether the computer is running.

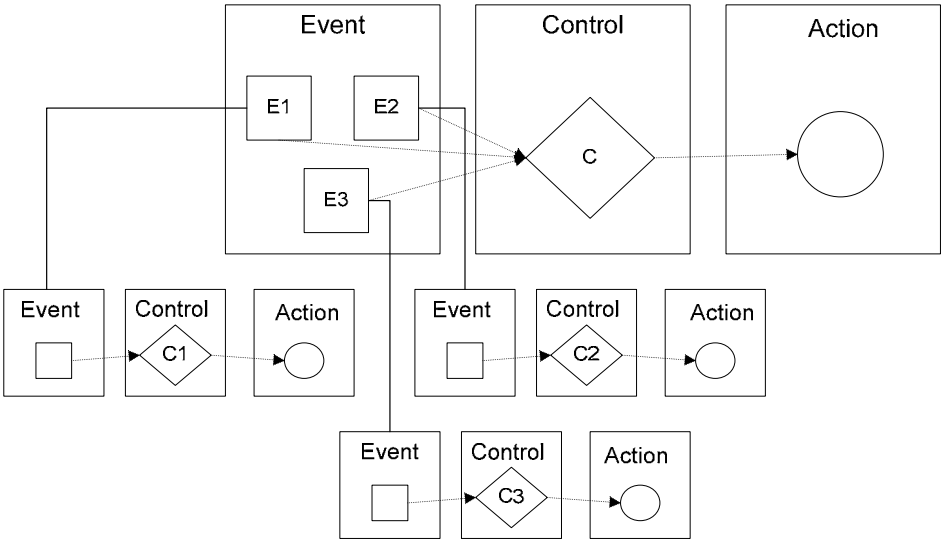

Figure 6 – Example of recursive application of the ECA pattern

The combination of events (Laura enters in Lab AND Laura is not with friends AND Laura's computer is on) is a compound event observed on the following components:

- A Context Manager component that detects whether Laura enters in Lab (E1 in Figure 6);

- A Context Manager component that detects whether Laura is not with friends (E2 in Figure 6);

- A Context Source component that detects whether Laura's computer is on (E3 in Figure 6);

Particularly, Context Manager E1 integrates the context information coming from two different Context Sources (e.g., a sensor for detecting that a person entered in the lab and Laura's recognition card for inferring that this person is Laura). The following ECA rule is described in Controller C1, characterizing the recursive nature of the ECA pattern:

If <Person enters in Lab AND this person is Laura> *then* <generate (Laura enters in Lab)>

Context Manager E2 integrates the context information that a person entered in the lab, coming from a Context Source, and the information that this person holds a recognition card. This means that this person is not a friend, since we are assuming, to simplify the reasoning, that each person who does not have a recognition card can be considered as a friend, namely as someone that is not in the lab for working. The following ECA rule can be described in Controller C2:

If <Person enters in Lab AND this person has a recognition card> *then* <generate (A person that is not a friend enters in the Lab)>

Finally, within Context Source E3, the following ECA rule is described in Controller C3:

If <Laura's computer is running> *then* <generate (Laura's computer is on)>

The hierarchy of Context Sources and Managers mentioned above is depicted in Figure 7.

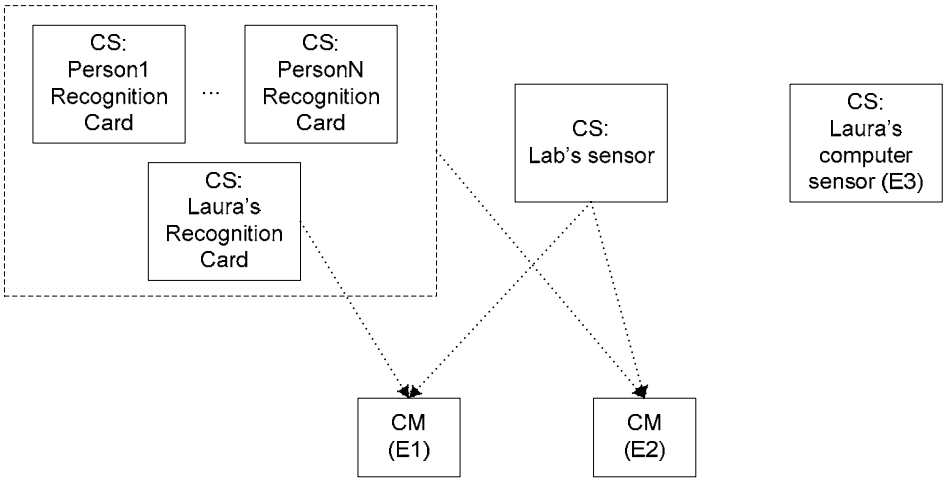

Figure 7 - Example of Context Sources and Managers hierarchy

2.4 Control Module

The *Control* module is responsible for observing context changes sensed by Context Source and Manager components (Event module), and, as consequence of these changes, to trigger actions that should be performed by the Action Performer component (Action module). This is possible by providing the Controller component that observes events and triggers actions with an application behavior description in which the reactive behaviors of context-aware applications are described in terms of condition (ECA) rules.

How to support the realization of the Control module for context-aware applications is the topic of this thesis. Figure 8 shows a possible architecture of the Control module, which consists of:

- A knowledge base (the application behavior description depicted in Figure 3) that describes what the context-aware application to be developed should do in terms of ECA rules, with the form *if* <*condition*> *then* <*action*>.

- A rule engine (the Controller component depicted in Figure 3) that should be able to process the knowledge base and execute the ECA rules. Particularly upon the occurrence of context changes sensed by an event source, the rule engine should have the capability to check the knowledge base for rules with an *if* portion that became valid due to these changes. For these rules, the rule engine should be able to execute the *then* portion with the help of an action performer.

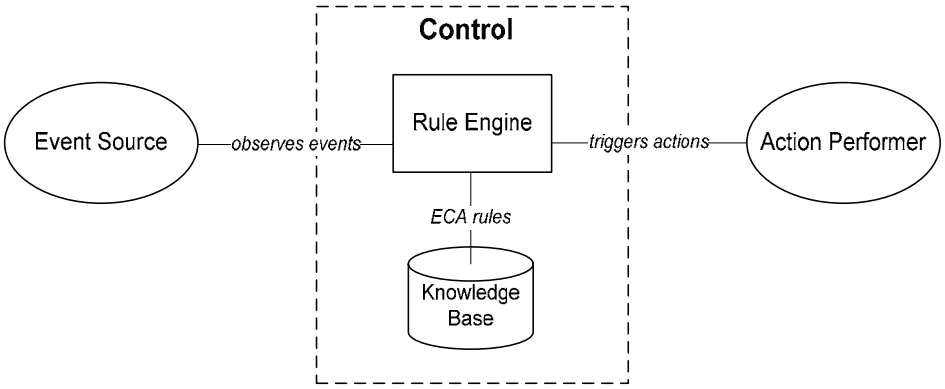

Figure 8 - Architecture of the Control module of the ECA pattern

The following chapters of this thesis discuss how a rule engine works, how it can process and execute ECA rules, and which specific rule engine to use for realizing the Control module.

2.5 Action Module

The *Action* module is responsible for performing the actions triggered by the Controller component. Actions represent an application reaction to context information changes, and these reactions may be the invocation of any internal or external service, such as the generation of a signal, the delivery of a notification or a web services request.

The Action module should enable coordination of actions and decoupling of action implementations from action purposes. An action purpose define an abstract action intention, while its implementation represents the realization of this intention using specific implementation technologies [25]. For this aim, we can divide the Action Performer, depicted in the right side of Figure 3, in three components, each of them with a specific task:

- An Action Resolver component that performs coordination of dependent actions. This component applies techniques to resolve compound actions, which are decomposed into indivisible units of action purposes (indivisible from the application point of view);

- An Action Provider component that defines action purposes, which describe an intention to perform an action with no indication on how and by whom these computations are implemented;

- An Action Implementor component that defines action implementations. This component defines various way of implementing a given action purpose. For example, the action “send notification” may have various implementations, each of them supported by a different service provider.

Figure 9 shows a class diagram of the Action module of the ECA pattern [25].

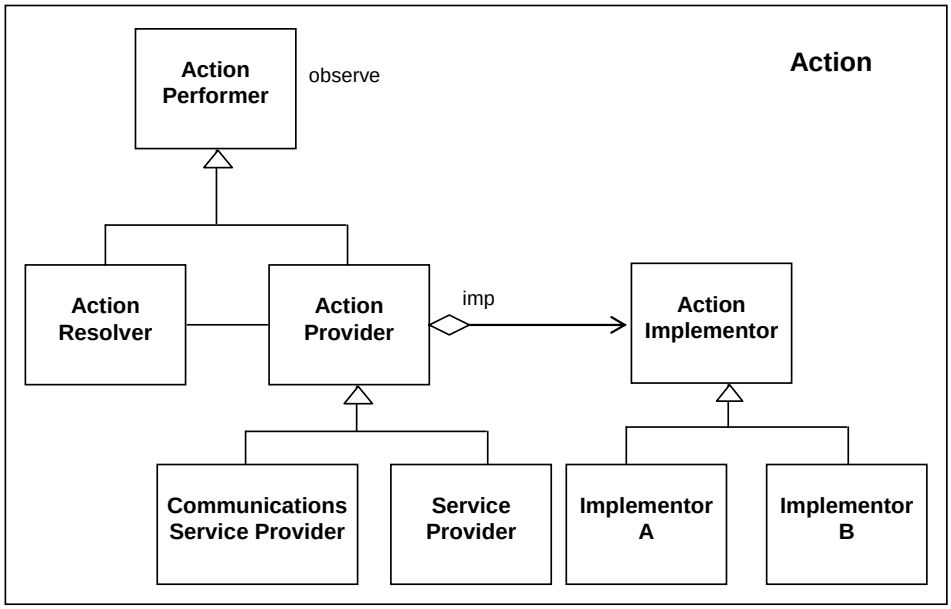

Figure 9 –Structure of the Action module of the ECA pattern

Both the Action Resolver and Action Provider components inherit the characteristics of the Action Performer component, and therefore they are both capable of performing actions.

The Action Resolver component performs compound actions, decomposing them into indivisible action purposes, which are further performed separately by the Action Provider component.

Action Providers may be communication service providers or (application) service providers. Communication service providers perform communication services, such as a network request, while service providers perform general application-oriented services, implemented either internally or externally to the application, such as a signal generation or a notification delivery, respectively.

An Action Provider may aggregate various Action Implementor components, which provide concrete implementations for a given action purpose. In Figure 9, two different concrete implementations are represented (Implementor A and Implementor B) [25].

2.6 Concluding Remarks

By applying the classic design principle of separation of concerns, the ECA architectural pattern effectively enables the distribution of responsibilities in context-aware applications. Context processor components encapsulate context related concerns, and actions are decoupled from control and context concerns, permitting them to be developed and operated either within or outside the application.

Applying such design principles greatly improves the extensibility and flexibility of a context-aware application, since context processors and action components can be developed and deployed on demand. Moreover, the definition of application behavior by means of ECA rules allows the dynamic deployment of context-aware applications.

The decomposition of Context Processor in a hierarchical structure of Context Source and Manager components enables a more effective, flexible and decoupled distribution of context processing activities (sensing, aggregating, inferring and predicting). This structure improves collaboration among context information owners and it is an appealing invitation for new parties to join this collaborative network, since collaboration among more partners enables availability of potentially richer context information.

Moreover, this decomposition enables filtering of unnecessary information across the hierarchy of context information processing units. At the lowest level of context information gathering, a great overhead of information flow can be detected but only the relevant information is kept and forward to the next level of the hierarchy.

The definition of a structure of Action Resolvers, Providers and Implementors enables the coordination of compound actions and the separation of abstract action purpose from its implementations. This structure avoids permanent binding between an action purpose and its implementations, allowing the selection of different implementations at runtime. In addition, abstract action purposes and concrete action implementations may be changed and extended independently, improving dynamic configuration and extensibility of the application [25].

3 ECA-DL

In Chapter 2 we have shown that reactive behaviors of context-aware applications can be expressed by using ECA rules. In this chapter we present a language that can be used to write ECA rules, coined ECA-DL. This language has been developed in the scope of the AWARENESS project.

The chapter is structured as follows: section 3.1 describes the basic concepts of ECA-DL; section 3.2 deals with the semantics and the syntax of the language, and, finally, section 3.3 presents examples of ECA-DL rules.

3.1 ECA-DL: a Rule Specification Language

ECA-DL is a *Domain-specific Language* specially developed for context-aware applications. For this reason, it is easier to use by context-aware application developers than general purpose languages.

Rules in ECA-DL are composed by an *Event* part that models an occurrence of interest in the context, a *Condition* part that specifies a condition that must hold prior to the execution of the action, and an *Action* part to be executed when conditions are fulfilled. Often the Action part of a rule consists of the invocation of a notification service, but it could also be any operation needed by the application. Observing this structure, we conclude that ECA-DL rules follow the ECA pattern, and can be used as means for writing ECA rules.

3.1.1 Basics concepts

ECA-DL has been developed with the following requirements in mind:

- Expressive power in order to permit the specification of complex event relations. Actually, ECA-DL allows the use of relational operator predicates (e.g., <, >, =), and the use of logical connectives (e.g., AND, OR, NOT) on events to build compound conditions;

- Convenient use in order to facilitate its utilization by context-aware application developers. Actually, ECA-DL provides high-level constructs that facilitate event compositions;

- Extensibility in order to allow extension of predicates to accommodate events being defined on demand, as well as event properties.

In the development of ECA-DL, a variant of the Situation-based triggering approach presented in [35] [36] has been adopted. In this approach, context changes are described as changes in situation states. *Situations* represent specific instances of context information, typically high level context information. Examples of possible situations of a person are isWorking, isOccupied or isReachable.

Situations may be defined upon other situations or facts. *Facts* define current "state of affairs" in the user's environment. Examples of facts are:

- Patricia is the supervisor of Laura;
- Laura is in the lab;
- Laura is with friends;
- Laura has computer on.

The concept of fact is too fine-grained to define context information at the level required by users and applications. There is a need for context abstractions capable of modeling context information that is closer to the matters perceived by users and their applications. To address this issue, situation context abstraction is used. The situation context abstraction allows application developers and users to leverage on the fact abstraction in order to derive high-level context information. For example, considering a context-aware scenario in which we want to monitor the working activities in a building of the University of Twente, a situation may be derived from facts as follows:

- the situation isWorking may be derived from the facts "a person is in the lab", NOT "this person is with friends", and "this person has computer on";

- the situations isOccupied and isReachable may be derived from the fact "there are available means of communication in the building".

Moreover, situations may be built upon other situations. For example:

- the situation isAvailable may be defined as not isOccupied and isReachable.

Facts and situations are defined as part of information models, which we have defined using UML as modeling language. Our models contain the definition of entities, context, and mutual relationships between each entity and its context. As is shown in Figure 10, any entity may be related to several different context types, as well as a specific context type may be referred to one or more entities.

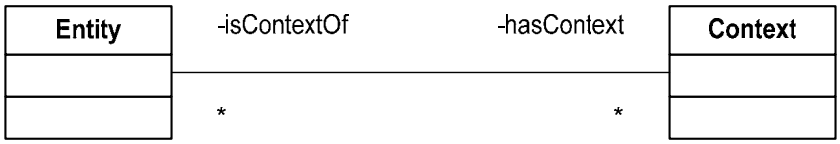

Figure 10 – General structure of an information model

3.1.2 ECA rules and ECA-DL rules

The structure of an ECA rule is depicted in Figure 11.

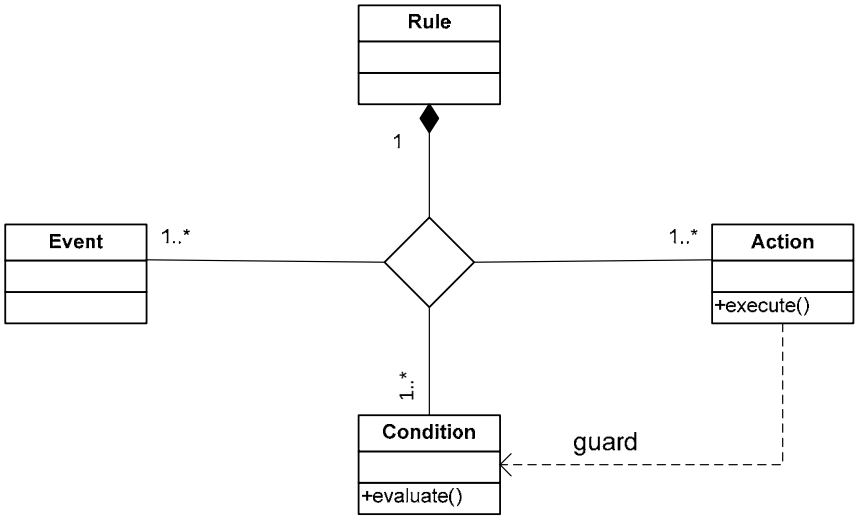

Figure 11 – Structure of ECA rules

An ECA rule consists of [26]:

- One or more *events*, which model the occurrence of relevant changes in our application or its environment (user's context). The occurrence of these events triggers the evaluation of the ECA rule.

- One or more *conditions*, which represent the situation under which the actions of the rule are enabled, in case the specified events occur. A condition is typically expressed as a (simple or complex) Boolean expression.

- One or more *actions*, which represent the operations of the rule that determine the reactive behavior of the application.

Events, conditions and actions may also have internal structure. For example, a condition may consist of multiple sub-clauses, or an action may be implemented as a procedure call that invokes several sub-procedures.

Based on the issues just discussed, we have identified the basic requirements for ECA-DL with respect to elements in the language. ECA-DL should offer the following:

- An *Event* part, which defines a relevant situation change in the context. Particularly, an event defines a basic change in the context for which the rule should be executed.

- A *Condition* part, which defines a logical expression that must hold following the event and prior to the execution of the action.

- An *Action* part, which represents the operation to be performed by the application as a consequence of the occurrence of a certain event and the fulfillment of the condition(s) associated with this event. An action often consists of the invocation of a notification service.

In ECA-DL rules, both events and conditions are defined in terms of situations and facts. Particularly, events in ECA-DL may have three possible states: *true*, *false* and *unknown*, and six state transitions between these states. The unknown state accommodates uncertainty of context information (when the value of context information is unknown). Figure 12 presents the state transitions possible for events given a certain situation.

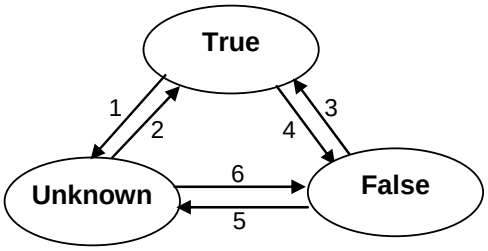

Figure 12 - State transitions for a situation

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

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

We consider only the true and false states and the possible transitions between these two states in this thesis. Since the development of ECA-DL is an ongoing work in the AWARENESS project, the unknown state is expected to be supported in the future.

3.2 ECA-DL: Semantics and Syntax

As depicted in the UML class diagram of Figure 13, a rule in ECA-DL consists of an Event part, a Condition part and Action part.

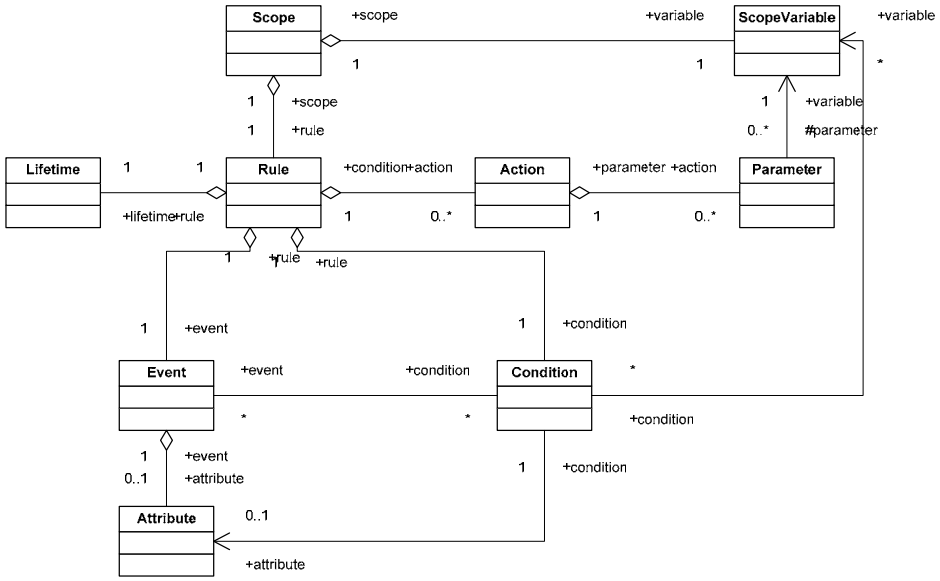

Figure 13 – ECA-DL language metamodel

The requirements presented in the previous section resulted in the clauses **Upon**, **When** and **Do**, respectively. Events are defined in the **Upon** clause, while conditions are specified in the **When** clause and, finally, actions are specified in the **Do** clause. In case there are no conditions to be specified, the **When** clause may be omitted.

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

In order to be able to filter collections for entities that fulfill a certain condition, we have defined the **Select** clause. It allows the selection of a subset of a collection respecting context and/or attributes constraints. For example, it may be necessary to select all users that are in the house and taking a shower, or we would like to select all devices that are currently being used, or even all the patients of the clinic that have diabetes.

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

Non-parameterized rules have the following basic structure:

```
Upon <event-expression>
When <condition-expression>
Do <notifications>
<lifetime>
```

Table 1 presents the semantics of the ECA-DL main clauses.

Table 1 - Semantics of the ECA-DL clauses

Clause	Semantics
Select	`Select (<collection-of-entities>, <var>, <filtering-expressions>)`

Upon-When-Do	`Upon <correlation-of-events>`
	`When <correlation-of-conditions>`
	`Do <notification>`

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

3.3 ECA-DL Rules: Examples

This section presents some examples in order to better understand how to define ECA rules in ECA-DL. For each example we propose:

- A context-aware scenario in natural language and the corresponding ECA rule;
- The corresponding information model used as basis for writing the rule;
- The ECA-DL version of the given rule.

3.3.1 Example 1

"Patricia would like to be notified when Laura enters the laboratory, without friends, and Laura's computer is on".

We can express this scenario by using the following ECA rule:

If <Laura is in the lab AND (NOT Laura is with friends) AND Laura's computer is on> *then* <Notify (Patricia), "Laura is working.">

The *if* part of the ECA rule consists of three situations and the *then* part consists of one action. The situation "Laura is in the lab" represents the event (the basic change in the context) for which the rule should be executed. Situations "NOT Laura is with friends" and "Laura's computer is on" represent additional conditions to be satisfied to enable the execution of the action.

The event “Laura is in the lab” is necessary to execute the *then* part, but by itself it is not enough. For example, Laura could be in the lab speaking with friends, in which case, she is not working. On the contrary, the aim of the rule is to notify Patricia that Laura is working, which is expressed by the additional conditions that Laura is not with friends and her computer is on.

Figure 14 depicts the information model used as basis to write the ECA rule of this scenario.

Error! Objects cannot be created from editing field codes.

Figure 14 – Information model of Example 1

As is shown in Figure 14, the entity Person is characterized by a name, and is associated through the relation hasLocation to the context Location, which can be decomposed in GeneralLocation (i.e., the building where the person is located), and SpecificLocation (i.e., the specific room of the building). In this example, the name of the person is Laura and she has specific location in Lab. A Friend is a person, and a person may be with one or more friends. Finally, a person may have one or more Computer, which may be running or not.

We can define the following ECA-DL rule for this example, which is consistent with the model in Figure 14:

```
Upon EnterTrue (Laura.inLab)
When (NOT (Laura.isWithFriends)) AND (Laura.hasComputerOn)
Do Notify (Patricia, “Laura is working.”)
Always
```

This ECA-DL rule is executed upon the event `EnterTrue (Laura.inLab)`and when both the additional conditions `NOT(Laura.isWithFriends)` and `Laura.hasComputerOn` are fulfilled. Lifetime **`Always`** is associated to this rule, and, therefore, this rule has no temporal constraints and should be always executed when both events and conditions are satisfied.

3.3.2 Example 2

“When Laura and Patricia start a meeting together, the meeting time should be counted”.

We can express this scenario by using the following ECA rule:

If <Laura is in meeting AND Patricia is in meeting AND Laura and Patricia share the meeting> *then* <Count meeting time>

The *if* part of the ECA rule consists of three situations and the *then* part consists of one action. Situations “Laura is in meeting” and “Patricia is in meeting” represent the event (the basic change in the context) for which the rule should be executed. The situation “Laura and Patricia share the meeting” represents the additional condition to be satisfied for executing the action.

The event “Laura is in meeting AND Patricia is in meeting” is necessary to execute the *then* part, but without the additional condition “Laura and Patricia share the meeting”, this event is not enough, since Laura and Patricia could be in different meetings.

Figure 15 depicts the information model that applies to this ECA rule.

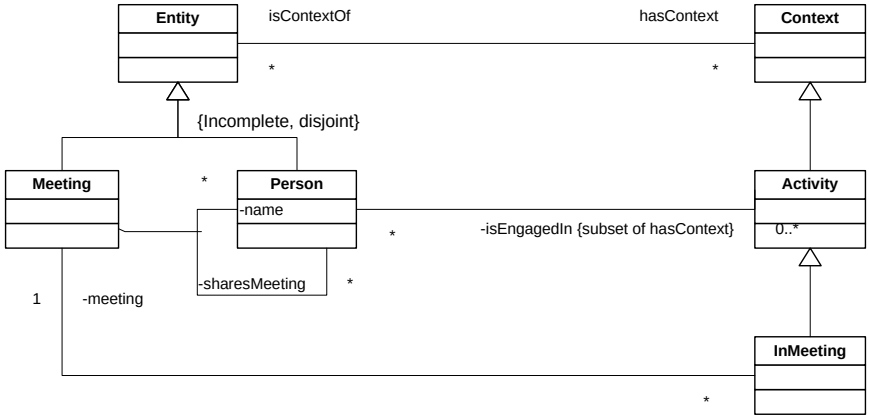

Figure 15 – Information model of Example 2

As is shown in Figure 15, the entity Person is characterized by a name, and is associated to the context Activity through the relation isEngagedIn. A person may be engaged in the activity inMeeting, which is associated to the entity Meeting. In this example, Laura is in meeting and Patricia is in meeting.

Moreover, a person may share a meeting with one or more persons. This explains why Meeting is defined as an association class relating two persons. In this example, Laura shares a meeting with Patricia.

Before writing the ECA-DL version of the rule, we decomposed the *then* part "Count meeting time" in two actions. For counting the meeting time, it is necessary to start counting the time when the meeting starts, and to stop counting the time when the meeting finishes. This means that in this scenario we need to write two ECA-DL rules: one for starting to count the meeting time when the meeting starts, and another one for stopping to count the meeting time when the meeting finishes.

Therefore, we obtained the following ECA-DL rules, which are consistent with the information model shown in Figure 15:

```
Upon EnterTrue (Laura.inMeeting)
     AND EnterTrue (Patricia.inMeeting)
When Laura.sharesMeeting (Patricia)
Do StartCountMeetingHours
Always

Upon TrueToFalse (Laura.inMeeting)
     OR TrueToFalse (Patricia.inMeeting)
When Laura.sharesMeeting (Patricia)
Do StopCountMeetingHours
Always
```

The first ECA-DL rule is executed upon events `EnterTrue (Laura.inMeeting)` and `EnterTrue(Patricia.inMeeting)`, and when the additional condition `Laura.sharesMeeting(Patricia)` is fulfilled.

The second ECA-DL rule is executed upon at least one of the events `TrueToFalse (Laura.inMeeting)`or `TrueToFalse(Patricia.inMeeting)`, and when the additional condition

`Laura.sharesMeeting(Patricia)` is fulfilled. Actually, the event that triggers the action `StopCountMeetingHours` is that Laura or Patricia are not in the meeting anymore (which means that the meeting has finished), but not in any meeting, since we refer to the meeting that Laura and Patricia share.

Both rules have lifetime **`Always`** and, therefore, they have no temporal constraints and should always be executed when both events and conditions are satisfied.

3.3.3 Example 3

"During the hot season, when the temperature in a building of the University of Twente is more than 30 degrees and it is later than 14:00 hours and earlier than 17:00 hours, all the persons in the building should be notified to go home".

We can express this scenario by using the following ECA rule:

If <During the hot season the temperature in a building of the University of Twente is more than 30 degrees AND it is later than 14:00 hours AND it is earlier than 17:00 hours > *then* <Notify (all the persons in the building), "You can go home.">

The *if* part of the ECA rule consists of four situations and the *then* part consists of one action. The situation "during the hot season" represents the lifetime associated with the rule. The situation "when the temperature in a building of the University of Twente is more than 30 degrees" represents the event (the basic change in the context) for which the rule should be executed. Situations "it is later than 14:00 hours" and "it is earlier than 17:00 hours" represent additional conditions to be satisfied to enable the execution of the action.

The event "the temperature in a building of the University of Twente is more than 30 degrees" is necessary to execute the *then* part, but by itself it is not enough. Actually, the aim of the rule is to notify all persons in the building to go home when the temperature inside the building is more than 30 degrees not in any time during the day, but just between 14:00 and 17:00 hours. Therefore, the rule needs the additional conditions "it is later than 14:00 hours" and "it is earlier than 17:00 hours" in order to be executed.

Figure 16 depicts the information model referenced by the ECA rule.

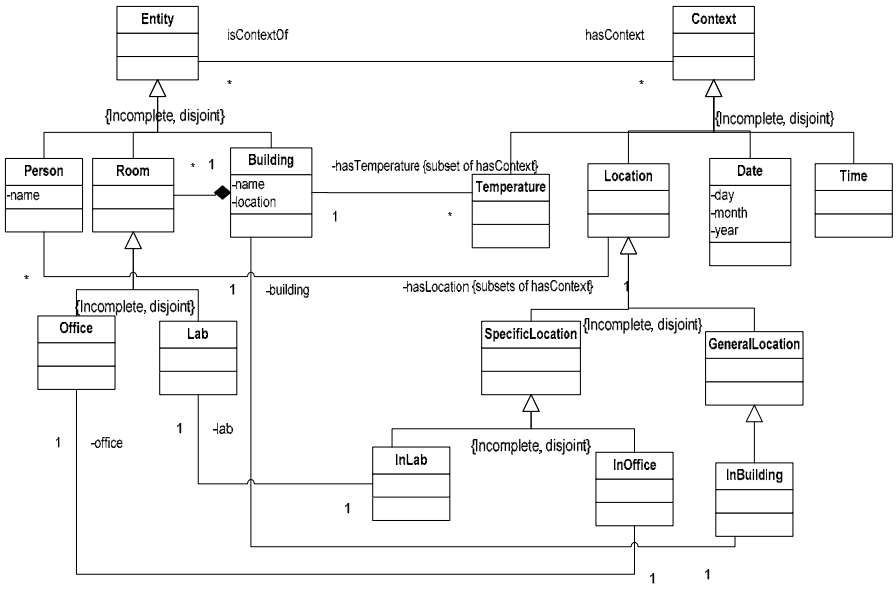

Figure 16 – Information model of Example 3

Similarly to the first example (see section 3.3.1), the entity Person is characterized by a name, and is associated through the relation hasLocation to the context Location, which can be decomposed in GeneralLocation (i.e., the building where the person is located), and SpecificLocation (i.e., the specific room of the building).

The entity Building is an aggregation of many rooms, for example, an office or a lab, and a building has a name and a location, which in our example is the University of Twente. Building is associated through the relation hasTemperature to the context Temperature in order to express the temperature in a building.

Finally, Figure 16 shows the context Time, which is necessary in order to know the current time, and the context Date, which has attributes day, month, year, and is necessary in order to express the lifetime associated with the rule.

We can define the following ECA-DL for this example, which is consistent with the model in Figure 16:

```
Scope (Select (building.*, build, build.inUT); b))
{
Upon EnterTrue (b.temperature > 30)
When (currentTime > 14) AND (currentTime < 17)
Do Notify (Select (person.*, p, p.InBuilding(b)),"You can go
home.")
from <May> to <September>
}
```

The **`Select`**`(building.*, build, build.inUT)` clause defines all building located in the University of Twente and the **`Scope`** clause stores this set of buildings in a variable `b`.

The rule is executed upon the event `EnterTrue (b.temperature>30)`, i.e., when the temperature in a building of the University of Twente is more than 30 degrees, and when the additional conditions `currentTime > 14` and `currentTime < 17` are fulfilled.

The **Do** clause selects all the persons in a building `b`, i.e., in the building of the University of Twente where the temperature is more than 30 degrees, in order to notify them to go home.

Finally, since the rule should be executed during the hot season, the lifetime associated with the rule is **from** <May> **to** <September>, which are the hottest months of the year.

3.3.4 Example 4

"All persons in the Zilverling building should be notified when there is a presentation in the building that is interesting for them".

We can express this scenario by using the following ECA rule:

If <There is a presentation in the Zilverling building AND there are persons in the building interested in this presentation> *then* <Notify (these persons), "This presentation may be interesting for you."><

The *if* part of the ECA rule consists of two situations and the *then* part consists of one action. The situation "there is a presentation in the Zilverling building" represents the event (the basic change in the context) for which the rule should be executed. The situation "there are persons in the building interested in this presentation" represents the additional condition to be satisfied for executing the action.

The event "there is a presentation in the Zilverling building" is necessary to execute the *then* part, but by itself it is not enough. Actually, when there is a presentation, the rule requires to notify persons in the building only if they are interested. Therefore, there is a need of the additional condition "there are persons in the building interested in this presentation" in order to execute the rule.

Figure 17 depicts the information model corresponding to the ECA rule.

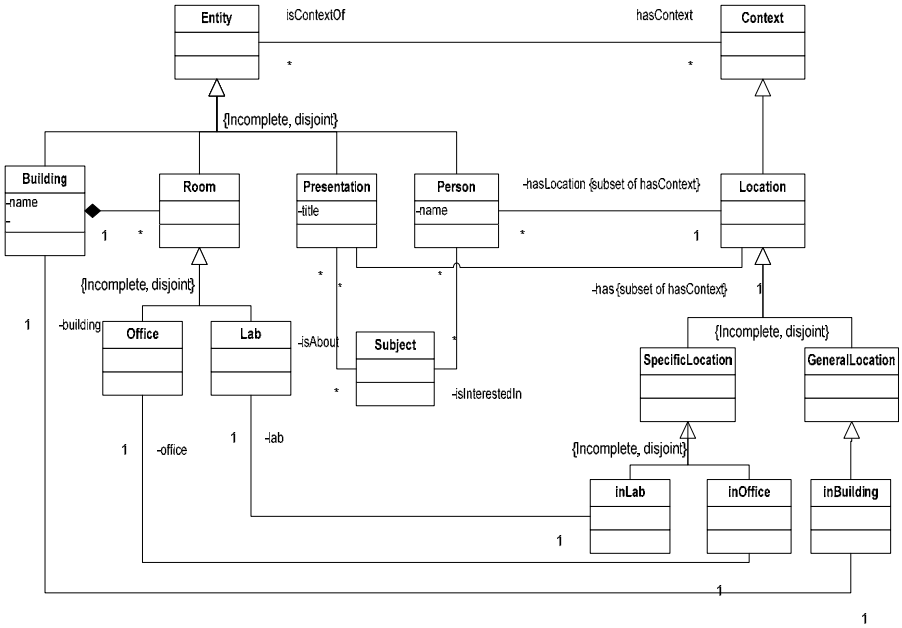

Figure 17 - Information model of Example 4

Similarly to the previous examples, the entity Person is characterized by a name, and is associated through the relation hasLocation to the context Location, which can be decomposed in GeneralLocation (i.e., the building where the person is located), and SpecificLocation (i.e., the specific room of the building). Moreover, in this example the entity Person may be interested in one or more subjects that are topics of one or more presentations.

The entity Presentation is characterized by a title, and is associated to the context Location, which can be decomposed in GeneralLocation (i.e., the building where the presentation is located), and SpecificLocation (i.e., the specific room of the building).

We can write the following ECA-DL version of the rule, which is consistent with the information model shown in Figure 17:

```
Scope (Select (persons.*, pers, pers.inBuilding.Zilverling);
p)
{
Upon EnterTrue (presentation.inBuilding.Zilverling)
When p.isInterestedIn(presentation.subject.*)
Do Notify(p),"This presentation may be interesting for you."
Always
}
```

The **`Select`** clause defines all persons that have general location `inBuilding`, more precisely in the building with name `Zilverling`. The **`Scope`** clause stores this set of persons in a variable `p`.

The rule is executed upon the event `EnterTrue (presentation.inBuilding.Zilverling)`, i.e., when there is a presentation in the Zilverling building, and when the additional condition `p.isInterestedIn(presentation.subject.*)`is fulfilled, i.e., when a person in the Zilverling building is interested in subjects that are topic of a presentation.

The **`Do`** clause notifies the set of persons `p` (defined in the **`Scope`** clause) that there is a presentation, which may interest them.

Finally, lifetime **`Always`** is associated to this rule and, therefore, this rule has no temporal constraints and should always be executed when both events and conditions are satisfied.

3.3.5 Example 5

"When a student is in a meeting with his/her supervisor(s), the meeting time should be counted".

We can express this scenario by using the following ECA rule:

If <Student is in meeting AND this student shares the meeting with his/her supervisor(s)> *then* <Count meeting time>

This rule is a generalization of the rule of Example 2 (see section 3.3.2). The *if* part of the ECA rule consists of two situations and the *then* part consists of one action. The situation "student is in meeting" represents the event (the basic change

in the context) for which the rule should be executed, while situation “this student shares the meeting with his/her supervisor(s)” represents the additional condition to be satisfied for executing the action.

The event “student is in meeting” is necessary to execute the *then* part, but by itself it is not enough. Actually, we are not interested in any meeting, but in the meeting shared between a student and his/her supervisor(s). Therefore, the rule needs the additional condition “this student shares the meeting with his/her supervisor(s)” in order to be executed.

Figure 18 depicts the information model corresponding to the ECA rule.

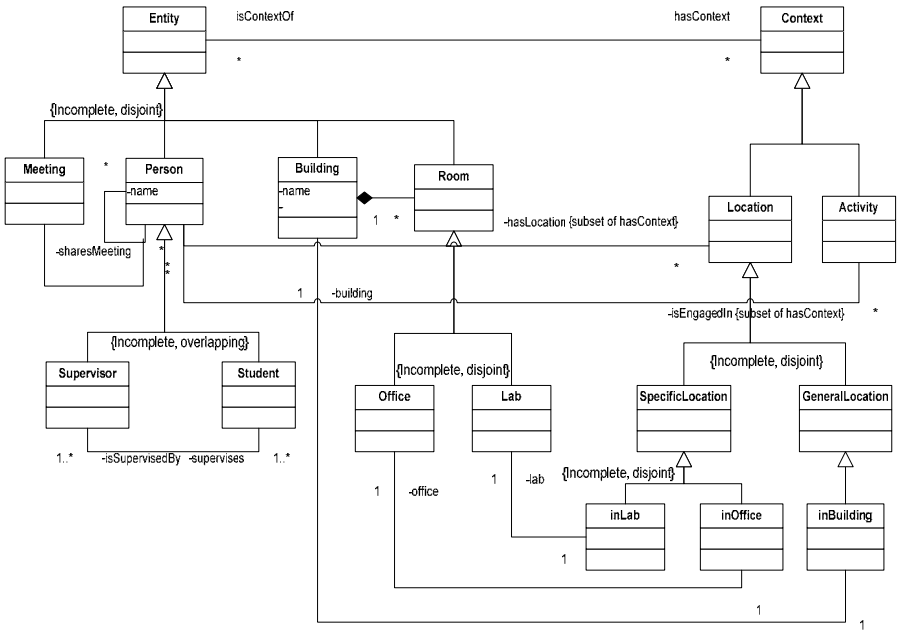

Figure 18 – Information model of Example 5

Similarly to the previous examples, the entity Person is characterized by a name, and is associated to the context Activity through the relation isEngagedIn. A person may be engaged in the activity inMeeting, which is associated to the entity Meeting. A person may share a meeting with one or more persons. This explains why Meeting has been defined as an association class between two persons. Moreover, Person is associated through the relation hasLocation to the context Location.

In addition to the other examples, a person may be Supervisor and/or Student. Actually, the same person may be a student but also a supervisor. A Supervisor supervises on or more students and a student may be supervised by one or more supervisors.

Similarly to the example in section 3.3.2, we decomposed the *then* part “Count meeting time” in two actions, one that starts counting the meeting time when the meeting starts, and another one that stops counting the meeting time when the meeting finishes. Therefore, we have defined the two following ECA-DL rules:

```
Scope (Select (student.*, st, st.inBuilding.Zilverling);
stud)
{
Upon EnterTrue (stud.inMeeting)
```

```
When stud.sharesMeeting (Select (supervisors.*, super,
superv.stud))
Do StartCountMeetingHours
Always
}

Scope (Select (student.*, st, st.inBuilding.Zilverling);
stud)
{
Upon TrueToFalse (stud.inMeeting)
When stud.sharesMeeting (Select (supervisors.*, super,
superv.stud))
Do StopCountMeetingHours
Always
}
```

In both the rules, the **Select**`(student.*, st, st.inBuilding.Zilverling)` clause defines all the students in the Zilverling building, and the **Scope** clause stores this set of students in a variable `stud`.

The first ECA-DL rule is executed upon the event `EnterTrue (stud.inMeeting)`, i.e., when a student of the Ziverling is in a meeting, and when the additional condition `stud.sharesMeeting(`**Select** `(supervisors.*, super, superv.stud))` is fulfilled, i.e., when a student of the Ziverling that is in meeting shares the meeting with his/her supervisor(s).

The second ECA-DL is executed upon the event `EnterTrue (stud.inMeeting)`, and when the additional condition `stud.sharesMeeting(`**Select** `(supervisors.*, super, superv.stud))` is fulfilled. Actually, the event that triggers the action `StopCountMeetingHours` is that the student is not in the meeting anymore (which means that the meeting has finished). This does not refer to any meeting, but specifically to the meeting that the student shares with his/her supervisor(s).

Finally, both the rules are associated to lifetime **Always** and, therefore, they have no temporal constraints and should always be executed when both events and conditions are satisfied.

4 Rule-Based Systems

This chapter presents an overview of expert systems based on rule engines and selects some criteria for comparing different existing tools for developing rule-based expert systems.

The chapter is structured as follows: section 4.1 deals with expert systems and their main features. Section 4.2 discusses rule-based systems. Particularly, it presents the basic model of rule-based languages, the general architecture of rule-based systems, the different kinds of strategies that they can use for processing rules, and, finally, the possible types of reasoning on which they are based. Section 4.3 gives guidelines useful for deciding when to use a rule-based system for developing a specific application. Finally, section 4.4 discusses the criteria we have defined to evaluate rule-based systems.

4.1 Expert Systems

Traditional programming languages, such as C, Pascal, Fortran and Cobol, are designed and optimized for the procedural manipulation of data [1]. They are based on the procedural programming paradigm that is characterized by the concept of procedure call. Procedures, also known as routines, subroutines, methods or functions, simply contain a series of computational steps to be carried out. Any given procedure can be called at any point during a program's execution.

Often these traditional languages are not suitable to solve complex problems of the real world. Actually, humans usually use abstraction and symbolism in order to facilitate the description of these problems. Explicative but not particularly detailed concepts are defined, and symbols that are appealing for human beings are used to represent these concepts. In Wikipedia [2], we can find the following definitions:

- *Abstraction*, in computer science, is a mechanism and practice to reduce and factor out details so that one can focus on few concepts at a time;

- *Symbolism* is the systematic or creative use of symbols as abstracted representation of concepts.

Although the human abstract and symbolic representation of concepts can be modeled in traditional procedural languages, considerable programming effort is required for that. This happens because procedural languages are sequential and deterministic, namely the programmer instructs a machine to perform some tasks in a specific order. Therefore, it is difficult and not straightforward to transform the human knowledge expressed at a high level of abstraction to a format suitable for procedural programming paradigms.

A branch of Artificial Intelligence tries to bridge the gap between human representation of knowledge and traditional programming languages. This branch concerns the development of techniques that allow the modeling of knowledge at a high level of abstraction. These techniques, which emulate human expertise in well defined problem domains, are characterized by the use of *expert systems* [1].

Expert systems are developed in languages and tools which are easier to use and maintain than traditional programming languages. These languages and tools allow building programs that closely resemble human logic in their implementation. Expert systems have been used to solve a wide range of problems in several domains, such as medicine, mathematics, engineering, geology, computer science, business, law, defense, and education.

Expert systems can be used cross-domains, since they can provide solutions to fundamental problems, independent on the domain. For example, considering diagnostics as a problem, we could implement a solution by exploiting the same expert system for medical diagnosis of a disease or for the diagnosis of a fault in a computer system, namely for two different domains [3].

Building an expert system is not a trivial task, since it is difficult to extract relevant knowledge from the problem domain and to express it in a proper manner for implementing a solution based on an expert system.

Generally, it is important to develop an initial prototype based on explanatory text and/or interviews with experts of the problem domain, in order to get more understanding of the domain. Further developments should refine this prototype, by testing and modifying it in collaboration with the experts and the potential users of the expert system.

4.2 Rule-Based Systems

Rule-based programming is one of the most commonly used techniques for developing expert systems. A rule-based expert system consists of a set of rules that can be repeatedly applied to a collection of facts. The following concepts are essential to rule-based systems:

- *Facts* represent circumstances that describe a certain situation in the real world.

- *Rules* represent heuristics that define a set of actions to be executed in a given situation.

There is a basic distinction between *derivation* and *production* rules. Derivation rules have the form *if* <*condition*> *then* <*conclusion*>, whereas production rules have the form *if* <*condition*> *then* <*action*>. The difference is subtle but flat. A conclusion, in derivation rules, is abstract: it consists of deriving logical

consequences from certain conditions. These logical consequences are simply asserted but not executed. An action, in production rules, is concrete: it consists of producing practical consequences from certain conditions. These practical consequences are concretely executed.

In this thesis, we are basically interested in production rules, since our aim consists of finding a rule engine able to process ECA rules, namely rules like *if* <*condition*> *then* <*action*>, which are production rules. However, production rules can implement derivation rules by using a special action "assert", which asserts knowledge. If the <*action*> part of a production rule is just a conclusion and not a function that performs actions, we can consider this production rule as a derivation rule. In both these cases, rules are composed of an *if* portion and a *then* portion.

Although tools for developing expert systems adopt different terminologies, we have chosen a specific terminology that we use consistently throughout this thesis. We have also related the terminologies adopted by different tools to our terminology in order to improve understandability for readers used to these tools. Our terminology is introduced below.

We call the *if* portion of a rule the *left hand side* (LHS), but sometimes it is called *predicates* or *premises*. The LHS consists of an expression, which can be a single expression (an *individual fact* that must be true for applying the rule) or a series of expressions (*composite expression*). In the literature of rule-based languages, a single expression is usually called *pattern*. A composite expression consists of several single expressions connected together by using the *conditional elements* "and, or, not" in order to create complex rules . Usually, when several expressions are connected by the "and" conditional element, this element is omitted. In rule-based languages we have also the *logical connectives* " &, |, ~", which are used to manipulate values inside a single fact . However, in this chapter we are interested in conditional elements "and, or, not", which are used to connect together several expressions, i.e., single expressions (facts) or composite expressions.

There is an essential difference between the concept of pattern introduced above, and the pattern concept discussed in Chapter 2 (ECA pattern). The ECA pattern refers to a *software architectural pattern*, which is used to describe a particular recurring design problem and present a generic scheme for its solutions. The term "pattern" in the sense of rules refers to the single expressions that compose the LHS of a rule.

We call the *then* portion of a rule the *right hand side* (RHS), but sometimes it is called *conclusions* or *actions* according to the type of rule (respectively, derivation or production rule). The RHS consists of a set of actions, represented by *functions*, to be executed when the rule is applicable [1]. The applicability of the rules depends on the method of reasoning (forward chaining or backward chaining, see section 4.2.4).

The general structure of a rule is the following:

If <(pattern1)…(patternN)> *then* <(action1)…(actionM)>

4.2.1 Basic model of rule-based languages

Figure 19 depicts the UML diagram representing the concepts of rule-based languages [28]. It consists of eleven classes: *Rule*, *LHS*, *Expression*, *Fact*, *Slot*,

CompositeExpression, *UnaryExpression*, *BinaryExpression*, *RHS*, *Action*, and *Function*.

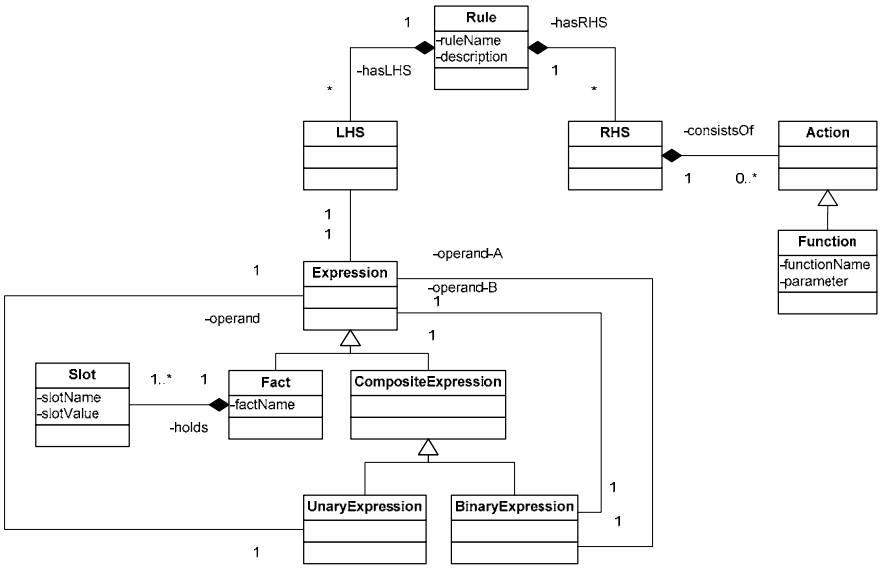

Figure 19 - Basic model of rule-based languages

Class Rule models rules. The ruleName attribute represents the name of the rule, and the description attribute specifies the purpose of the rule. Since a rule consists of a LHS (left hand side) and a RHS (right hand side), the class Rule has two references: hasLHS and hasRHS.

The LHS of a rule is an expression, which can be a Fact or a CompositeExpression.

A Fact is a single expression that must be true for applying the rule. A fact is characterized by a name and a collection of slots. Therefore, class Fact has a factName attribute and a containment reference to Slot. Class Slot has two attributes, slotName and slotValue, which respectively represents the name of the slot and its value.

A CompositeExpression consists of several expressions connected together by using the *conditional elements* "and, or, not". Particularly, a CompositeExpression can be a BinaryExpression by using "and, or" conditional elements, or a UnaryExpression by using "not". A BinaryExpression consists of two operands, A and B, while an UnaryExpression consists of only one operand. These operands are expressions, which can be a fact (the final element) or a composite expression again. For example, consider the following rule:

If <AND((fact1) (fact2))> *then* <(action1)>

<AND((fact1) (fact2))> represents the LHS, which is an expression. This expression is a CompositeExpression, more specifically a BinaryExpression, whose operands are fact1 (operand-A) and fact2 (operand-B). Each of these operands are again expressions. In this case, they are facts, which are the final elements in the "recursion chain". However, these operands could be expressions again. For example, consider this rule:

If <AND((OR(fact1)(fact2)) (fact3))> *then* <(action1)>

<AND((OR(fact1)(fact2)) (fact3))> represents the LHS, which is an expression. This expression is a BinaryExpression with an operand-A, which is OR(fact1)(fact2), and an operand-B, which is fact3. Each of these operands are again expressions, however, in this case, operand-A is again a BinaryExpression composed by two operands (fact1, fact2) connected by using the "or" conditional element.

Class RHS has a containment reference Action. The RHS of a rule consists of zero or more actions. An action is a function characterized by a name and parameters, which represent the arguments that we can pass to the function. Therefore, class Function has two attribute, functionName and parameter.

4.2.2 Rule-based systems architecture

The general architecture for rule-based systems is depicted in Figure 20.

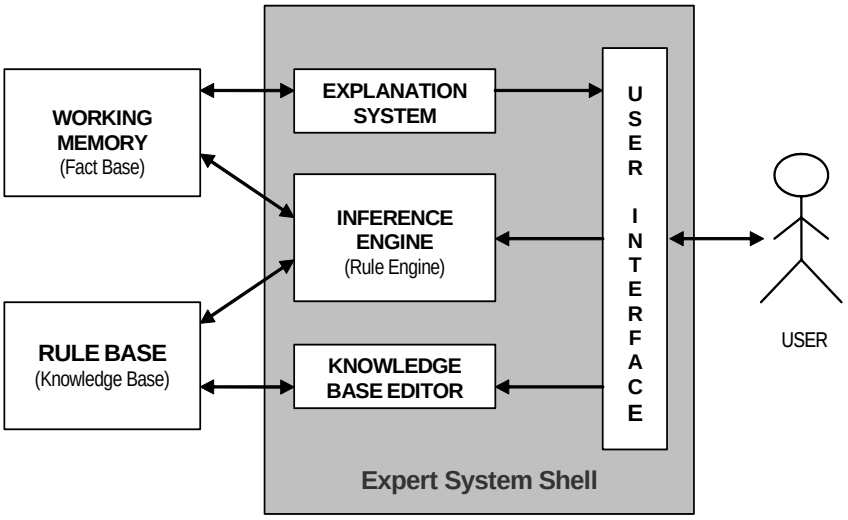

Figure 20 - General rule-based systems architecture

The main elements of a rule-based system are facts, rules, and the engine that acts on them. The core of the architecture shown in Figure 20 consists of the *working memory* (fact base), the *rule base* (knowledge base) and the *inference engine* (rule engine).

- The working memory contains facts that are the smallest piece of information supported by the rule engine.

- The rule base contains rules in the form of *if-then* statements, which represent the knowledge provided by the user and/or an expert of the problem domain;

- The inference engine matches facts in the working memory against rules in the rule base, and it determines which rules are applicable according to the reasoning method adopted by the engine. The list of applicable rules in the inference engine is usually called *conflict set*. When instructed to begin execution, the inference engine selects a rule from the conflict set and fires this rule by executing the associated actions. The rule to be selected depends on the specific conflict strategy used by the system. After that, the inference engine selects another rule and repeats the process until no applicable rules remain in the knowledge base.

The other components shown in Figure 20 are the *user*, the *user interface*, the *explanation system* and the *knowledge base editor*.

The user interacts with the system through a user interface that may use menus, natural language or any other style of interaction. Almost all expert systems have an explanation system that allows the system to explain its reasoning to the user. Some systems have a knowledge base editor, which helps updating and checking the knowledge base.

The inference engine, the user interface, the explanation system and the knowledge base editor constitute the *expert system shell*. In contrast, the user, the fact base and the knowledge base, which are domain-specific knowledge, are not considered as part of this shell. Given a specific problem domain, we can usually find an expert system shell that provides support for it, since there are numerous commercial expert system shells suitable for a different range of problems. In this case, all we need to do is to provide the system with a fact base and a knowledge base encoded in the form of rules. The usage of expert system shells generally reduces the cost and time of development (compared with writing the expert system from scratch) [4]. Most modern rule engines can be seen as more or less specialized expert system shells, with features to support operation in specific environments or programming in specific domains.

4.2.3 Conflict resolution strategies

When the inference engine matches facts in the working memory against rules in the rule base, there is a need to choose which of the applicable rules is fired first. This is a task of the conflict resolution strategy adopted by the specific system. This strategy depends on the kind of problem to solve and/or on the importance that the user gives to each rule. There are several conflict resolution strategies, but the most common are the following [23]:

- *First-applicable*. If the applicable rules contained in the conflict set are in a specific order, the simplest strategy consists of firing the first rule in the list. The drawback of this strategy is that we could have an infinite loop on the same rule. Actually, if the working memory does not change over time, facts matching the first rule are always the same, and this rule is fired again and again. For solving this problem, it is a common practice to suspend a fired rule until working memory changes occur on the facts that mach this rule.

- *Random*. A random strategy simply chooses a single random rule to fire from the conflict set. The drawback is that this strategy does not provide the predictability or control of the first-applicable strategy, but it is suitable for systems that require unpredictability, such as games.

- *Most-specific*. This strategy is based on the number of *patterns* in the *if* part of the rules. The approach consists of firing the rule with the highest number of patterns matched. The strategy is based on the assumption that the rule with the highest number of patterns is the most relevant to the contents of the working memory.

- *Least-recently-used*. In this strategy each rule is accompanied by a time or step stamp, which marks the last time that the rule has been used. In this strategy, the rule to be fired first is the least recently used one, i.e., the one with the oldest time or step stamp. This strategy maximizes the number of individual rules that

are fired at least once, improving the fairness of the rules firing. This strategy is suitable when all rules are needed for solving a given problem.

➢ *Salience*. This strategy gives a weight, called *salience*, to each rule. The salience is a number that specifies the priority of a rule. The rule with the highest salience is fired first.

4.2.4 Forward chaining and backward chaining

Rule-based systems may support two different types of reasoning: *forward chaining* and *backward chaining*. They may use either or both methods, but forward chaining is the most common one.

Forward chaining is often called data-driven, in contrast to backward chaining, which is referred to as goal-driven. For understanding the difference, we can think about the ways in which a detective might solve a mystery. Suppose he has a collection of evidences, like a fingerprint, a dead body, etc. The first possibility consists of drawing conclusions from the available evidence, adding them to the available information, and continuing until a link between evidence and crime is found [17]. This is the forward chaining method and it is shown in Figure 21 [23].

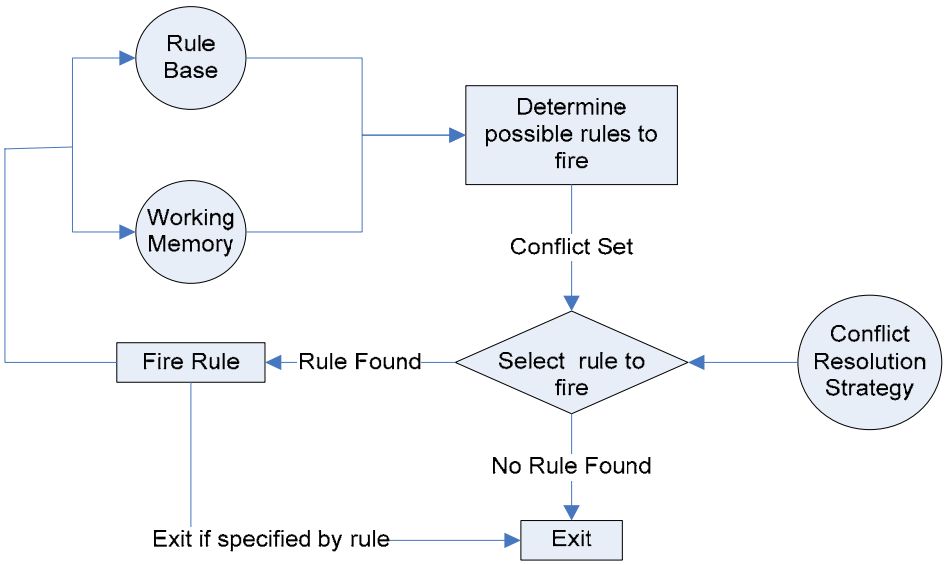

Figure 21 - Forward chaining reasoning

The forward chaining strategy starts with the available data, represented by facts in the working memory, and uses the rules to extract more data until a desired goal is reached. This goal could be, for example, the firing of a certain rule. As depicted in Figure 21, a rule-based system using forward chaining matches facts in the working memory against rules in the rule base until it finds rules where the *if* portion is known to be true. After deciding which rule to fire (according to the conflict strategy), the system can conclude the *then* portion, which may result in the addition of new facts to its datasets. The rule-based system often cycles through this process until a desired goal is reached.

For example, suppose we have the following available data [29]:

I have a pet named Fritz and he hops. This knowledge is represented by the facts "Fritz is a pet" and "Fritz hops" in the working memory. Suppose also we have defined the following rules in the rule base:

1. If Fritz hops → then Fritz is green

2. If Fritz is green → then Fritz is a frog

The *if* portion of rule 1 is known to be true (given the knowledge in the working memory), therefore the system can conclude the *then* portion, and the new fact "Frits is green" is added to the working memory. As consequence, the *if* portion of rule 2 is known to be true (given the actual knowledge in the working memory) and the new fact "Fritz is a frog" can be added to the expert system datasets.

Consider again the example of the mystery to solve. Instead of using a forward chaining reasoning, the detective can proceed in an alternative way. He can start from the circumstances of the crime, form a hypothesis about what happened, and then search for clues that support this hypothesis [17]. This technique is an example of backward chaining reasoning and it is shown in Figure 22 [23].

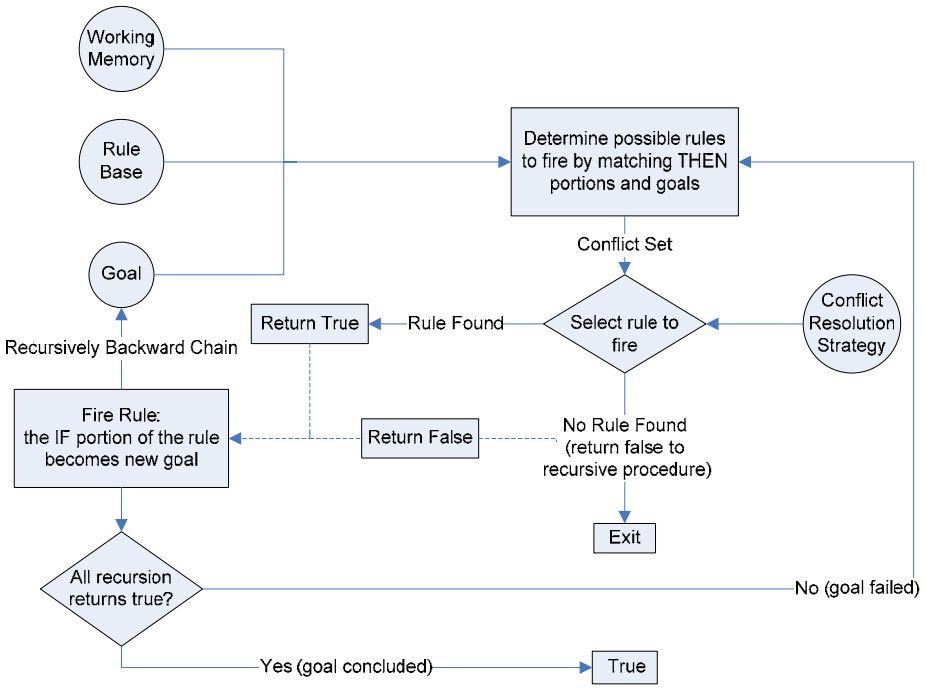

Figure 22 - Backward chaining reasoning

The backward chaining strategy starts with a list of goals and determines the conflict set by searching for rules with a *then* portion that matches the goals. According to the conflict resolution strategy, the first of these rules is fired and the *if* portion of this rule becomes a new goal for the system. This backward chain continues recursively until either every available data satisfies the goal (goal concluded), or there are no more rules that match the goal (goal failed).

For example, consider these rules [30]:

1. If Fritz hops → then Fritz is green

2. If Fritz is green → then Fritz is a frog

The final goal is to conclude that Fritz is a frog given he hops. When using backward chaining, rule 2 is selected to be executed since its *then* portion matches the goal (Fritz is a frog). It is not yet known whether Fritz is green, and this way, the *if* portion of rule 2 becomes a new goal (Fritz is green). Rule 1 is now selected to be executed, since its *then* portion matches the new goal (Fritz is green). The *if* portion of rule 1 (Fritz hops) is known to be true as part of the first goal. This way, the initial goal can be concluded, i.e., that Fritz is a frog.

We can use forward chaining with both derivation and production rules, while backward chaining can only be used for derivation rules. Actually, in order to apply backward chaining in production rules, the rule engine should execute the *then* portion that matches a desired goal and reason backwards. This is not realistic and efficient, since the *then* portion of a production rule is an action to be concretely executed and not a conclusion that needs only to be asserted in the working memory, like in derivation rules.

4.3 When to Use a Rule-Based System

Rule-based systems are suitable for some kinds of problems, and less suitable for others. Therefore, it is important to propose guidelines for deciding whether to use rule-based solutions in a project. The following aspects should be considered when choosing a rule-based system [5]:

- *Type of algorithm*. When the algorithm that implements the solution of the problem is computing-intensive or a table-lookup and does not involve much conditional branching or decision-making, is not advisable to use a rule engine. In this case the problem might be easily solved using traditional computing methods.

- *Rules complexity*. In a rule-based expert system it is necessary to describe the problem through rules, by specifying the decisions that need to be made. In case it is possible to describe the problem in terms of a collection of complex rules, the usage of a rule engine is justified. This means that we should have rules composed of many conditions, such as, for instance, a block with three or more nested if-statements in pseudo-code, in order to consider the use of a rule-based solution.

- *Changes over time*. Another decisive requirement for choosing rule-based systems is the problem structure. If the problem is well-structured and basically static, it probably does not need the flexibility of a rule engine. However, if it has dynamic features and needs changes over time, we can be sure that a rule engine is suitable for tackling this problem.

- *Code maintenance*. The effort required for developing a solution and the aim of the project are also important aspects to consider. Thus, if the code developed during the project and the final product are going to be maintained over time, the effort involved in using a rule engine is justified. This effort concerns all the resources used in the project, such as time (training developers and possibly end-users) and money (licensing fees for using a commercially available engine, or investments for developing and deploying a new engine).

- *Performance optimization*. In case it is necessary to hard-wire the algorithm for absolute high-end performance, such as to optimize for speed and for memory, it is better to choose another alternative than rule engines. Actually, rule-based solutions are normally slower than their hard-wired counterparts.

4.4 Evaluation Criteria for Rule-Based Systems

The aim of this work is to develop an expert system able to notice changes occurring in a specific user's context, and to perform proper actions as consequences to these changes. Particularly, the aim is to develop a rule-based system able to understand and process context-information expressed in terms of ECA rules. Therefore, we need to search for a rule-based tool that is suitable for our needs.

There are many existing tools with similar features, but also some differences. These differences allow us to prefer a rule engine above another. In fact, the arduous task is not to choose between the rule engines that we have selected, since each of them is a powerful development tool, which offers a wide range of functionality. The problem is how to be sure that we are making the best choice for our needs. Therefore, our approach has been to collect at first meaningful and relevant criteria for selecting a rule engine that is suitable for our purpose.

This section describes these criteria. For each criterion we introduce both a general definition [2] and an explanation of the reason why this criterion is important in the selection process. In Chapter 5 we analyze several rule-based systems based on these criteria in order to choose the best option among the available ones.

4.4.1 Knowledge representation

Knowledge representation is needed for processing concepts in an information system. Concepts are abstract and universal notions that serve to designate a category of entities, events or relations between them. An information system is a technologically implemented medium for recording, storing and disseminating linguistic expressions, as well as for drawing conclusions from such expressions. In the field of Artificial Intelligence, for example, problem solving can be facilitated by an appropriate choice of knowledge representation.

Several programming languages have been developed that are oriented to knowledge representation. The most important in the scope of this thesis are rule-based and declarative languages. Rule-based languages allow us to write programs consisting of a collection of rules. We discuss the basic model of rule-based languages in section 4.2.1. Analogously, declarative languages allow us to write programs consisting of a list of rules describing data properties and feasible transformations of these data.

The development tool to be chosen should be capable of reasoning using rule-based languages or declarative languages, which are both suitable for knowledge representation based on rules.

4.4.2 Portability

Software portability consists of the capability of some software to be adapted or modified in order to be used in a different computing environment than the one for which this software was originally written. Porting is usually required because of differences in the CPU, operating system interfaces (APIs), hardware, or because of subtle incompatibilities in the programming language used in the target environment.

Portability is important since the code implementing our system could function also in another platform (CPU + architecture + operating system), if necessary, without any modifications. Portability allows the re-use of implementations and this implies the reduction of the software development costs.

4.4.3 Integration/Extensibility

The software development process often implies collaboration between members of a team, and each of these members may have his own preference in the language to use for working on his task. Moreover, sometimes a specific language may be more suitable than others for implementing certain functionality of the system. Therefore, integration may be needed in case of large systems, since modules developed using different programming languages may have to be integrated. Integration may also be necessary in case existing modules have to be re-used in a system.

Extensibility measures how difficult it is to extend a system and the effort required for implementing this extension. Extension means addition of new functionality or modification of already existing functionality.

The rule engine to be chosen should support a high level of integration and extensibility, since we may need to integrate the specific rule-based language of the engine with code written in another programming language. Furthermore, we may need to extend the functionality of the engine with newly defined requirements, which have not been foreseen at system design time.

4.4.4 Tools support

A programming tool is a program or an application used by software developers for creating, debugging, or maintaining other programs or applications. Some tools have been integrated in more powerful development environments (Integrated Development Environments, IDEs), which usually consist of a source code editor, a compiler and/or interpreter, build automation (for automatic recompilation), and a debugger.

The tools and IDEs supported by the rule engine to be chosen are certainly an important criterion in order to increase the productivity in software development. Actually, they provide additional help to the developers, above all in object-oriented programming, like:

- code completion verification;
- suggestion of the parameters to be passed to methods;
- direct access to a Concurrent Version System (CVS), which keeps track of all versions and all changes in the implementation of a software project, and allows several, potentially widely separated, developers to collaborate;
- refactoring functionality, which consists of automatic code re-writing, e.g., in case classes are changed.

4.4.5 *Expressiveness*

Language expressiveness represents the easiness with which the solution of a certain problem can be expressed using a language. The speed of problem solving with a specific language, the amount of code needed and even the quality of the final result are factors that depend on expressiveness. In general terms, when a language allows writing algorithms with few instructions and in a clear and readable way, it holds good expressiveness.

Expressiveness is difficult to evaluate since it is an intrinsic characteristic of the language, and it is also relative (it depends on who evaluates the language) . A way to evaluate the expressiveness of the several languages associated with the engines is to consider the logic on which they are based. Rule-based languages usually use first-order logic, which is a symbolized theory in which each sentence, or statement, is broken down into a subject and a predicate. The predicate modifies or defines the properties of the subject, and can only refer to a single subject. We expect for our engine a level of expressiveness at least comparable to the level held by the first-order logic.

4.4.6 *Alignment with ECA rules*

The Controller component we have designed senses events (changes occurrence) in a specific context. When an event is notified, the component checks the rules for true conditions and, for these rules, performs proper actions. This process is based on ECA rules with a general form*: if <condition> then <action>*. Our purpose has been to develop the Controller component using an existing rule-based system that holds its own language to express rules. Therefore, a decisive criterion for choosing an engine in our work has been whether the engine can give support to ECA rules.

4.4.7 *Commercial aspects*

Whether the software to be chosen is of public domain or available at no costs for academic use has certainly played a fundamental role in our choice. Moreover, there should be a community of users for discussing research, development and implementation of the software considered. Actually, such a community provides a basic contribution to the improvement and the evolution of these tools, and allows their users to find answers to any questions in addition to useful applications examples.

5 Rule Engines

This chapter presents some available rule-based engines and discusses them on the light of the criteria defined in the previous chapter.

Sections 5.1 to 5.4 present the rule engines CLIPS [14], Jess [15], jDREW [16], and Mandarax [17] respectively, focusing on their main features and feasible applications, and giving examples of the supported languages. Finally, section 5.5 compares these engines in order to justify our final choice.

5.1 CLIPS

CLIPS (C Language Integrated Production System) is a production development and delivery expert system tool written in C, which provides a complete environment for the construction of rule and/or object based expert systems [13].

5.1.1 Main features

CLIPS provides a cohesive tool for handling a wide variety of knowledge with support for three different programming paradigms: rule-based, object-oriented and procedural.

CLIPS is written in C for portability and speed, and runs on many different operating systems. Operating systems on which CLIPS has been tested include Windows 2000/XP, MacOS X, and Unix. It can be ported to any system which has an ANSI compliant C or C++ compiler, and comes with all source code, which can be modified or tailored to meet a user's specific needs.

5.1.2 Application environment

CLIPS code can be used in different ways:

- it can be embedded within procedural code and called as a subroutine. This allows the easy integration of CLIPS with existing systems and is useful in case the expert system is a small part of a larger task or needs to share data with other functions.

- it can be integrated with languages such as C, Java, FORTRAN and ADA.

This addition of external functions allows CLIPS to be extended or customized in many different ways.

A CLIPS expert system may be executed in three ways: interactively using a simple, text-oriented, command prompt interface; interactively using a window/menu/mouse interface on certain machines; or as an embedded expert system in which the user provides a main program and controls the execution of the expert system [15].

The standard version of CLIPS provides an interactive, text-oriented development environment, including debugging aids, on-line help, and an integrated editor. Interfaces providing features such as pull-down menus, integrated editors, and multiple windows have been developed for the MacOS, Windows 95/98/NT, and X-Window environments.

5.1.3 CLIPS language

Fact is one of the basic high-level concepts for representing information in a CLIPS system. Each fact represents a piece of information that has been placed in the current list of facts, called the fact-list. A fact may be added to the fact-list (using the `assert` command), removed from the fact-list (using the `retract` command), and modified (using the `modify` command) [15]. In the following example we add the fact "stoplights", which holds the value "red", to the fact-list using the command assert.

```
(assert (stoplights red))
```

A fact is the fundamental unit of data used by rules. A rule is one of the primary methods for representing knowledge in CLIPS and it defines a set of actions to be performed in a given situation. CLIPS supports the forward chaining strategy, which means that the left-hand-side (LHS) of the rule is a set of conditions that must be satisfied for the rule to be applicable. The conditions of a rule are satisfied based on the existence or non-existence of specified facts in the fact-list. The right-hand-side (RHS) of the rule is the set of actions to be executed when the rule is applicable, namely when the LHS of the rule is satisfied. An example of a rule in CLIPS is:

```
(defrule RULE "example of rule's syntax"
(stoplights red) => (assert (action StopTheCar))
```

CLIPS uses the `defrule` command for defining a rule. The name of the rule and a comment between quotes follow the `defrule` command. When the LHS part (stoplights red) is satisfied, the action StopTheCar is executed.

As we can see, CLIPS can be used to support ECA rules. Actually, the LHS part of a CLIPS rule can be thought as the *if* <*condition*> part of an ECA rule, and the RHS part as the *then* <*action*> part. In the example above the corresponding ECA rule could be:

If <Stoplights is red> then <Stop the car>

5.1.4 Applicability

CLIPS expressiveness permits the specification of very complex relations within the rules. Nevertheless, the overuse of parenthesis in the CLIPS language, and the need to use inverse polish notation for building arithmetic and conditional expressions, make it an inconvenient language for the programmer.

The CLIPS inference engine implements the RETE algorithm [1], which is a very efficient algorithm to solve the difficult many-to-many matching problem. CLIPS has attempted to implement this algorithm in a manner that combines efficient performance with powerful features.

CLIPS is maintained as public domain software.

5.2 Jess

Jess (Java Expert System Shell) is a fast and powerful rule engine for the Java platform, which supports the development of rule-based systems that can be tightly coupled to code written entirely in Java. Jess is also a powerful Java scripting environment, from which it is possible to create Java objects, call Java methods, and implement Java interfaces without compiling any Java code. Jess was originally conceived as a Java clone of CLIPS, but nowadays it has many features that differentiate it from CLIPS [16].

5.2.1 Main features

As well as in CLIPS, there are three ways to represent knowledge in Jess: rule-based, object-oriented and procedural programming paradigms. We can develop software using only rules, only objects, or a mixture of rules and objects. Furthermore, Java software built using Jess has the capability of reasoning using knowledge supplied in the form of declarative rules.

Jess is probably the most flexible rule engine on the market, since it has been used in different environment ranging from Windows CE handhelds to full-blown J2EE enterprise applications. Licensed users get the source code, so users may modify the engine if they find it necessary.

5.2.2 Application environment

Jess has been integrated with agent frameworks and other tools. It is also been integrated with the popular ontology editor Protégé 2000 [31]. This is a powerful combination that many people use to develop knowledge structures as well as code that acts on them. Moreover, Jess supports a whole range of different rule languages. For example, in the latest version of Jess there is a native XML rule language support.

Jess provides an editor with code completion and syntax checking as you type, a debugger, an explorer that lets you probe the relationship between templates and rules, and a rule database.

A fundamental feature for our project is that the latest release of Jess includes an Eclipse-based development environment (the JessDE). Eclipse [32] is an emergent and popular Open Source Integrated Development Environment (IDE), which

combines the open source philosophy with an open, extensible framework, and encourages the creation of a community of people to extend the capabilities of the IDE, allowing various different languages and applications to be supported by the environment.

An important requirement for us is to be able to extend the standard functionality of our rule engine (by using Java), in order to allow it to process ECA rules expressed in the ECA-DL language. This task can be facilitated using the JessDE, since it is a suitable and well supported environment for developing Jess code and integrate it with some other Java code.

5.2.3 Jess language

A program written in Jess may consist of facts, rules and objects. Facts represent all the pieces of information the rules work with. As in CLIPS, rules have two parts: a left hand side (LHS) and right hand side (RHS). The LHS is strictly defined for matching fact patterns. The RHS defines a list of actions to be performed if the pattern(s) of the LHS is (are) satisfied. Actions are typically method calls. Two additional capabilities of Jess are that the LHS can contain patterns that match external Java objects, and the RHS can call not only native Jess methods, but instance methods of externally referenced Java objects and static class methods. Via Java JNI, we could even call functions in other languages like C.

The Jess syntax is similar to the CLIPS syntax. However, there are some features that make the Jess language easier and more flexible. The general form of a Jess rule is:

```
(defrule RuleName “comment”
(fact_1). . . (fact_N) =>(action_1) . . . (action_M) )
```

This rule can be easily mapped on a corresponding ECA rule. The LHS portion of the Jess rule can be seen as the *if*<*condition*> part of an ECA rule, and the RHS as the *then* <*action*> part.

5.2.4 Applicability

Jess supports both forward and backward chaining, but Jess’s backward chaining version is not transparent to the programmer. Actually, it is necessary to declare the specific kinds of facts that can serve as backward chaining triggers, and only specific rules have to be defined to be used in backward chaining. In fact, Jess’s reasoning engine is strictly a forward chaining engine, and, therefore, backward chaining is effectively simulated in terms of forward chaining rules. However, the simulation is quite effective, and Jess’s backward chaining mechanism has many useful applications [17].

As CLIPS, Jess uses the RETE algorithm [33] to process rules. However, Jess has enhanced and refined the algorithm to improve the performance and flexibility of the system.

Jess can be licensed for commercial use, and is available at no cost for academic use.

5.3 jDREW

jDREW (Java Deductive Reasoning Engine for the Web) is a configurable and powerful deductive reasoning engine written in Java and well integrated with the Web.

5.3.1 Main features

Knowledge-based systems to process the declarative information and rules can use jDREW as an embedded reasoning engine through its various application programmer's interfaces (APIs). jDREW can be easily deployed as part of a larger Java system, on a server or, with its small memory footprint, on a client [18].

jDREW was designed to be flexible also in its capabilities and it provides modules to process rules in Prolog and Rule Markup language (RuleML) format. jDREW is part of the Rule Markup Initiative [34] that has defined a shared Rule Markup Language (RuleML), permitting to use both forward and backward reasoning on rules written in XML for deduction, rewriting, and further inferential-transformational tasks.

The goal of the Rule Markup Initiative is to develop RuleML as the canonical Web language for rules using XML markup, having formal semantics, and allowing efficient implementations. RuleML covers the entire rule spectrum, from derivation rules to transformation rules and to reaction (production) rules. RuleML allows one to specify queries and inferences in Web ontologies, mappings between Web ontologies, and dynamic Web behaviors of workflows, services, and agents.

5.3.2 The RuleML Initiative: general architecture of rules

The RuleML Initiative has developed a modular RuleML specification and transformations from and to other rule standards/systems. Moreover, it coordinates the development of tools to elicit, maintain, and execute RuleML rules. Figure 23 shows the hierarchy of rules adopted by the RuleML Initiative.

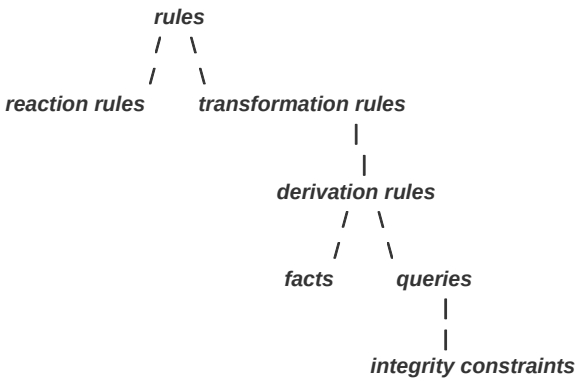

Figure 23 - The RuleML hierarchy of rules

The RuleML hierarchy of rules is a reduction tree rooted in general rules. Its main branches distinguish reaction rules and transformation rules. Directly below transformation rules are derivation rules. Derivation rules specialize to facts and queries, which themselves can become integrity constraints.

These different types of rules and how jDREW executes them can be explained in the following ways [19]:

- General rules are captured at high level in the form of natural language statements. In certain cases, these statements may be transformed into executable rule expressions by using formal notations, preferably declarative rule languages. In other cases, these statements cannot or do not need to be transformed. General rules may be also stated in a combination of both natural language and declarative executable languages [36].

- Reaction rules are production rules (see section 4.2), which are concerned with the invocation of actions in response to events and state the conditions under which actions must be taken. Reaction rules have an operational semantics with no clear logical semantics. Therefore, they can be processed just using the forward reasoning in a natural fashion, like in deductive databases, observing/checking events/conditions and performing an action if and when all events/conditions have been recognized/fulfilled. In our project, we are interested in this kind of rules, since they have a direct correspondence with ECA rules.

- Transformation rules are general rules whose event trigger is always activated. For transformation rules the backward reasoning is normally preferred.

- Derivation rules (see section 4.2) consist of knowledge statements that are derived from other knowledge by an inference calculation. Derivation rules have a logical semantics that allow them to be processed using both a forward reasoning and a backward reasoning. Since in different situations different kinds of reasoning for derivation rules may be optimal (forward, backward, or mixed), RuleML does not prescribe any one of these.

- Facts are derivation rules that have a positive (true) conjunction of *premises* in the LHS. For facts or *unit clauses* there is no notion of reasoning strategy.

- Queries are derivation rules that have a negative (false) disjunction of *conclusions* in the RHS. Queries may be proved by using both forward and backward reasoning.

- Integrity constraints are usually processed by using the forward reasoning, i.e., triggered by update events, mainly for efficiency reasons. However, they can be processed instead by using the backward reasoning, trying to show (in)consistency by fulfilling certain conditions (without need for recognizing any event).

5.3.3 Applicability

jDREW is an engine for clausal first-order logic. The jDREW APIs makes use of powerful deduction techniques like, for example, discrimination trees that identify the choices of what clauses can extend a given goal. jDREW does this efficiently in that it executes independently of the number of non-matching clauses [18].

In its current state, jDREW forms a component suite that consists of various software modules (components). Each component carries out an individual function of the inference procedure and can be easily assembled into a reasoning system.

jDREW is an open source project under the GNU General Public License (GPL).

5.4 Mandarax

Mandarax is an open source Java class library for deduction rules. It provides representation, persistence, exchange, management and processing (querying) of rule bases. The main objective of Mandarax is to provide a pure object-oriented platform for rule-based systems [21].

5.4.1 Application environment

The main application of Mandarax is the query-driven model based on derivation rules. Therefore Mandarax has been designed to process data held in external information sources that are usually huge relational databases. This is an important distinction between Mandarax and rule engines as CLIPS and Jess. CLIPS and Jess (and the rule-based systems in general) load facts from the knowledge base into their working memory in order to process the rules. In contrast, Mandarax allows the easy integration of all kinds of external data sources with no need to load this information into the working memory. For instance, database records can be easily integrated as sets of facts, and reflection is used in order to integrate functionality available in the object model. Other data sources (EJB beans, Web services, etc.) can be integrated as well.

Rule bases can be made persistent using the XKB module [20]. This module stores rules and other knowledge in a format similar to RuleML. The Mandarax team itself is part of the RuleML initiative working on a XML standard for rules [34]. Export and import of RuleML rule bases is supported by Mandarax. The RuleML editor is a Mandarax-based application to edit and query knowledge bases stored as RuleML XML files.

Extensions to Mandarax are available, including graphical user interface components (Swing and servlet/JSP tag based) and other add-ons [20].

5.4.2 General architecture

We illustrate the Mandarax mechanisms in Figure 24 with a rule-based query in a Client/Server architecture, which is a typical application of Mandarax [22]. In Figure 24 we show the interactions between client and server. The client communicates with the server using HTTP requests. The server processes them and sends method calls held in the requests to the logic layer, where the Mandarax service is situated. This service does not have a working memory to load the knowledge base, but it is a service able to directly interact via SQL queries with a persistency layer consisting of a server database. The results of these queries are processed by the Mandarax service and then delivered to the client via HTTP responses.

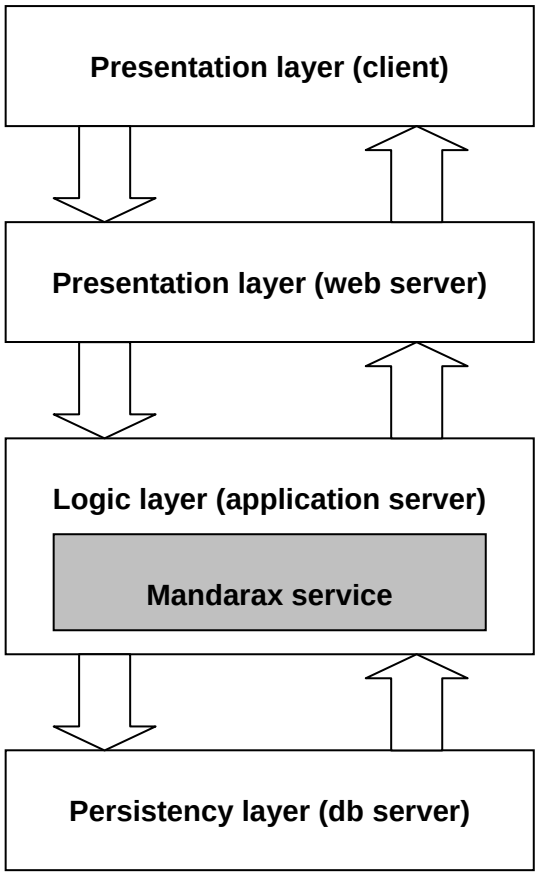

Figure 24 - Rule-based query answering in Client/Server architecture

5.4.3 Mandarax language

In Mandarax, Java interfaces and classes represent the various elements in logical expressions. Consider the following example:

If the turnover of a customer is higher than 500$ *then* the customer qualifies for a discount of 5%.

This derivation rule shows that one (or many) conditions can be associated with one conclusion. Sometimes the conditions are also called prerequisites or *body* of the rule, while the conclusion is called its *head*. The meaning of a rule is simple: whenever all conditions are true, the conclusion is true as well. Usually the conditions of a rule are connected using “and”, meaning that all conditions must be satisfied. Conditions can also be connected by “or”, meaning that the conclusion is true if at least one of the conditions is true.

The conditions and the conclusion are facts, which can also occur in isolation. The facts themselves consist of terms and predicates associating these terms. According to object-oriented terminology, terms represent objects, and predicates represent relationships between terms. In the example above, “higher than” and “qualifies for” are both predicates, while “500 $”, “ 5%”, “ customer” and “the turnover of a customer” are terms. There are three different kinds of terms: constant terms, variable terms and complex terms. Constant terms are more or less concrete objects like “500$” or “5 %”. In contrast, “customer” is a variable, which

is a kind of placeholder that can be replaced by concrete terms if needed. This makes sense in particular for rules, since otherwise we would need to define a separate rule for each customer. Finally, complex terms can be computed from other terms. In the example, "the turnover of a customer" is a complex term. The function "turnover of a customer" is an instruction on how to build a new term from two other terms. The terms inside a complex term can again be constant, variable or complex. Figure 25 shows the detailed relation between logic concepts in Mandarax and object oriented concepts in Java [22].

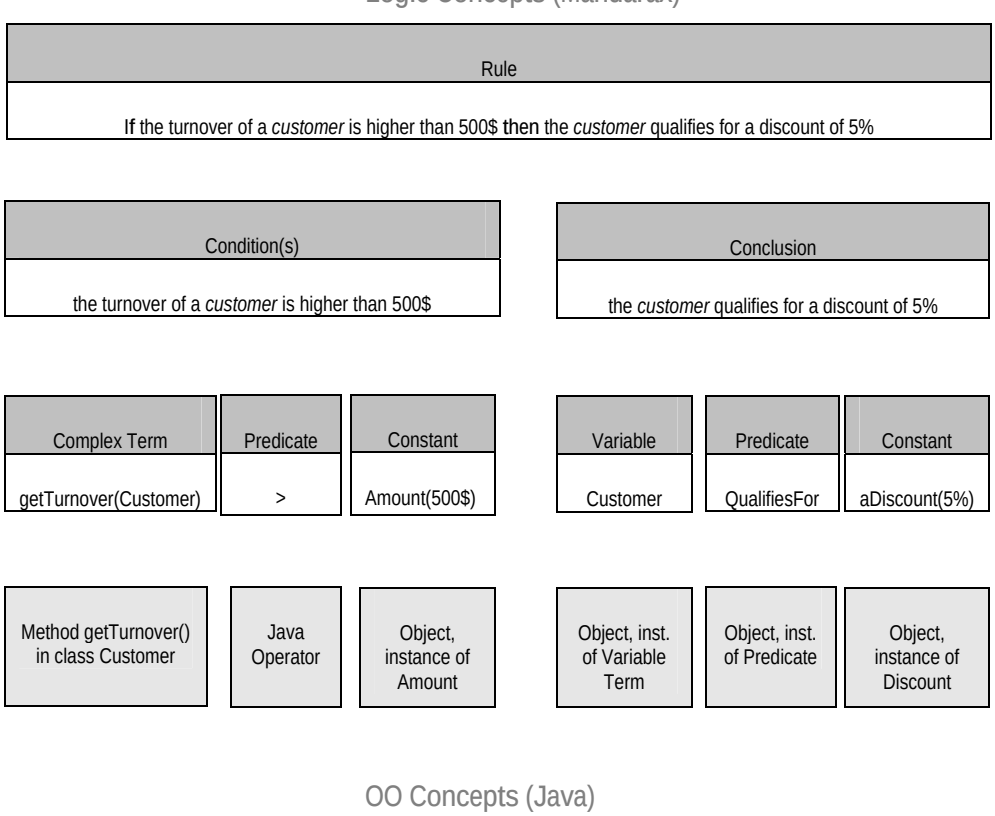

Figure 25 – Mandarax concepts and their corresponding Java programming concepts

5.4.4 Mandarax in a context-aware scenario

Considering the alignment with ECA rules, the Mandarax website [20] states that it provides a Mandarax ECA, which is an extension that can be used to program reactive agents. The system is event-driven: after events have registered event listeners (handlers), these listeners query the knowledge base for the next action that must be performed. Both the event and the action mechanism are designed for distributed systems. Although Mandarax ECA could be useful for our project due to the support it offers to ECA rules, documentation on Mandarax ECA is hard to find and for this reason we could not understand how this mechanism works. Therefore, we could not use Mandarax ECA and we can conclude that, at the moment, Mandarax does not directly support knowledge originated from event generators (event-based systems).

We discuss the support that Mandarax provides to ECA rules with an example depicted in Figure 26. Since Mandarax is based on querying model as opposed to

an events model, context-aware scenarios characterized by events-based models are not directly supported by Mandarax.

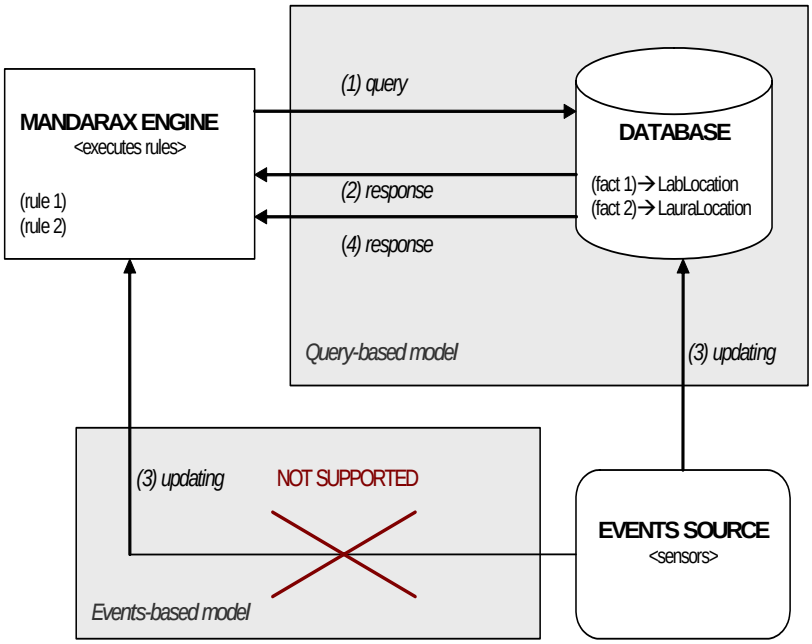

Figure 26 - Mandarax application in context-aware scenario

We consider the following context-aware scenario:

- In an external database there are two facts:

    1. The location of Laura's laboratory (LabLocation), which is static information that does not change over time.

    2. Laura's location (LauraLocation), which is dynamic information that changes over time.

- In the Mandarax engine we may have two simple rules associated with these facts:

    1. *If* (Laura is in the Lab) *then* (Send to her supervisor the notification "Laura is working");

    2. *If* (Laura is not in the Lab) *then* (Send to her supervisor the notification "Laura is not working").

The Mandarax engine continuously queries the database for checking if Laura is in the laboratory (message 1 in the picture). We suppose that initially Laura is in the laboratory. Therefore the initial response to the Mandarax engine is "Laura is in the Lab" (message 2 in the picture). Mandarax executes the RHS of rule1 and sends to the supervisor the "Laura is working" notification.

If at a certain moment Laura leaves the laboratory, the events source senses this change and automatically updates the database. Therefore, the response to the query (1) becomes "Laura is not in the Lab" (message 4 in the picture). Mandarax

executes the RHS of rule 2 and sends to the supervisor the notification “Laura is not working”.

As we can see in Figure 26, Mandarax does not support the direct updating of the rule engine, which would be desirable in a context-aware scenario.

5.4.5 Applicability

The Mandarax inference engine uses (query-driven) backward reasoning, and the reference implementation uses an object-oriented version of backward reasoning similar to the algorithm used in Prolog. Prolog is a logic programming language with simple and clear syntax and semantics, which is used in many artificial intelligence programs.

The Mandarax inference engine is very flexible: unification algorithm, loop checking algorithm and selection policy can be configured. In contrast, most commercial rule systems such as popular open source solutions, like CLIPS and Jess, use forward reasoning, in particular the RETE algorithm. The Mandarax approach is more appropriate in a query-driven (HTTP requests, SQL queries) system environment, where facts can stay in the database and are integrated on the fly. However, if the rule base (not the underlying database) is large, forward reasoning systems provide much better performance. Users reported that the Mandarax API is much simpler to use than Jess. This topic has been discussed in various discussion groups and opinions are controversial [20].

Mandarax is free and open source. The software license used is the GNU lesser general public license, making the software suitable for both open-source and commercial projects.

5.5 Rule engines’ comparison

In Table 2, we compare CLIPS, Jess, jDREW and Mandarax, in order to decide which tool is the most suitable for our purposes.

Each engine has benefits and drawbacks, and choosing the best one depends on the application being developed. Most engines we have studied seem to give support to several aspects we presented in section 4.4. Considering we are interested in developing context-aware applications, some engines perform better than others. The following paragraphs analyze these engines on the light of the criteria we have defined.

We assign the marks “+” or “++” to the rule engines when the result of the current evaluation is good or excellent, respectively. For some feature, the symbol “•” means that the engine supports this feature. For example, in the case of the portability, it means that the tool can be re-used in several operating systems, and, in the case of the commercial aspects, that the tool is free, open source and has a users’ community.

Table 2 - Comparison between rule engines

	CLIPS	JESS	JDREW	MANDARAX
Knowledge representation	++	++	+	+
Portability	•	•	•	•
Integration/Extensibility	+	++	+	+
Tools support	+	++	+	+
Expressiveness	++	++	++	++
Alignment with ECA rules	++	++	+	+
Commercial aspects	•	•	•	•

CLIPS and Jess can handle a wide variety of knowledge with support for rule-based, object-oriented and procedural programming paradigms, while jDREW and Mandarax can just process knowledge in the form of deduction/declarative rules. For this reason the evaluation of the knowledge representation supported by CLIPS and Jess is excellent (++), whereas for jDREW and Mandarax it is just good (+). This criterion is important for our choice, since we are interested in developing our rule-based system using an engine that gives support to the object-oriented programming paradigm.

All the rule engines support integration and hold a sufficient level of extensibility, but we evaluate Jess as being better than the others because of the wide capabilities to extend its functionality and its flexibility to be integrated with many other languages and systems.

Tools support is the decisive criterion for choosing Jess. Each engine provides interactive development environments, but Jess comes with JessDE, an Eclipse-based development environment.

Expressiveness is also an important criterion, since it denotes the concepts we can express using the basic constructs of each specific language. All the engines hold an excellent level of expressiveness, which permits the specification of very complex rules.

Concerning language and its ease of use, CLIPS language is considered as inconvenient for the programmer because of the overuse of parenthesis and the need to use inverse polish notation for building arithmetic and conditional expressions. Moreover, concerning the logic and the reasoning support, Jess and jDREW provide both forward chaining and backward chaining strategies to process the rules, while CLIPS rules can be used only for forward reasoning and Mandarax rules only for backward reasoning.

Concerning the alignment with ECA rules, Mandarax scores low since it is suitable for working with derivation rules and remote information sources, like

huge databases of bank systems, but it is not the best option for dealing with production/ECA rules and event-based systems.

jDREW scores very low if we consider the availability of documentation. It has been difficult for us to understand how this rule engine works. On the contrary, CLIPS, Jess and Mandarax have full and exhaustive documentation available on the Web. We think this difference has become evident by considering the presentations of the rule engines (see sections 5.1 to 5.4). In section 5.3, we have reported about the hierarchy of rules that the jDREW engine can process, but we could not find any explanation for its internal mechanisms nor application examples.

We conclude that Jess is the most suitable rule engine for our work, since it supports all the features we need for designing and implementing our Controller component in the scope of the AWARENESS project.

6 Jess

The goal of this chapter is to introduce Jess and its basic functions, and to discuss how to write simple applications in Jess.

The chapter is structured as follows: section 6.1 discusses the Jess architecture and section 6.2 describes examples of Jess applications. Section 6.3 consists of an introduction to the Jess language. Section 6.4 describes how to represent the different kinds of facts supported by Jess, while section 6.5 details how to write rules. Section 6.6 discusses the most powerful features of Jess, namely those that facilitate its integration with Java. This section explains how to embed Jess in Java applications. Finally, section 6.7 discusses the use of Jess in context-aware applications.

6.1 The Jess Architecture

Since we have decided to use Jess in order to design our context-aware Controller component, the first step towards this goal is to present the Jess architecture.

We have already introduced the basic components of a rule engine (see section 4.2.2), which are the working memory, the rule base, and the inference engine. Figure 27 shows these components in the Jess architecture, underlining their inner structure and their mutual connections.

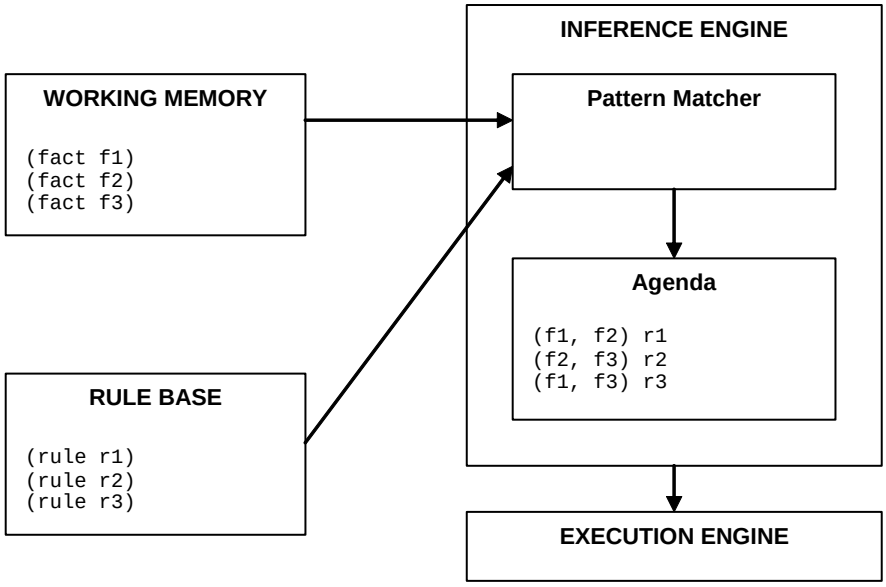

Figure 27 - The Jess architecture

The Jess rule engine does not contain any facts or rules until they are loaded, respectively, into the working memory and rule base.

The *working memory* contains facts and for this reason it is also called a *fact base*. Facts are all pieces of information the Jess rule engine works with, and they can be used as both LHS and RHS of the rules. The Jess working memory is similar to a relational database, where facts are like rows of the database maintained with indexes to speed up searching in the working memory.

The *rule base* contains all the rules that the engine knows. In rule-based systems, rules are sometimes stored as strings of text, but Jess holds a rule compiler for processing them into some format that the inference engine can manage more efficiently. Particularly, the Jess's rule compiler builds a *Rete Network*, which is a complex and indexed data structure for speeding up rule processing.

The *inference engine* decides what rules to fire and when. It consists of the *pattern matcher* and the *agenda*.

The *pattern matcher* decides which rules to activate based on the current contents of the working memory. A rule is activated when the pattern matcher finds facts that can satisfy the RHS of this rule. This is not a trivial task if we take into account that the working memory may contain thousand of facts and the rule base contains complicated rules with several premises and conclusions. In these cases, the pattern matcher might need to search through millions of combination of facts to find those combinations that satisfy rules. Fortunately, efficient ways exist to solve the problem, since a lot of research has focused on this area.

The *agenda* stores the list of rules that could be potentially fired. The agenda consists of an ordered list of rules, whose RHS can be executed. The agenda has to decide which rules have the highest priority and should be fired first. This process is called *conflict resolution strategy* and usually it takes into account:

- the specificity or complexity of each rule;
- the relative age of the LHS of each rule in the working memory.

We discuss conflict resolution strategies adopted by rule-based systems in section 4.2.3. Jess allows its users to set the conflict resolution strategy, particularly, the *least-recently-used* (default strategy) and the *first-applicable*. Moreover, Jess allows the programmer to set a specific priority for the rules (*salience* strategy), so that certain more important rules are always fired first.

Finally, the *execution engine* fires the rules. It is the component that executes the RHS of the rule once the inference engine decided what rule to fire. Jess offers a complete programming language to define what happens when a given rule fires, whereas other rule-based systems just offer the possibility of adding, removing, and modifying facts in the working memory.

The Jess rule engine works in discrete cycles consisting of three steps, as we can see in Figure 27:

1. All the rules contained into the rule base are compared to the working memory in order to decide which ones should be activated during this cycle. The list of these activated rules, together with any other rules activated in previous cycles, is called the *conflict set*.

2. The conflict set is ordered to form the agenda. This process is called *conflict resolution*. The used strategy depends on many factors, some of them under the programmer's control.

3. To complete the cycle, the first rule on the agenda is fired and the entire process is repeated again.

6.2 Jess Applications

We present below an overview of the Jess application areas and we discuss the possible ways to use Jess in combination with Java. This analysis is useful for understanding how we are going to exploit the functionality of the Jess engine and its flexibility.

Jess has been used to develop a broad range of commercial software, including:

- expert systems that evaluate insurance claims and mortgage applications;
- agents that predict stock prices and buy and sell securities;
- network intrusion detectors and security auditors;
- design assistants that help mechanical engineers;
- smart network switches for telecommunications;
- servers to execute business rules;
- intelligent e-commerce sites;
- games.

When the aim is to develop large applications, Jess can be used in two different but overlapping ways: as a tool for developing rule-based systems and as a general-purpose programming language.

So far, we have talked about Jess as a rule engine that can continually apply hundreds, or even thousand, of rules to a set of data. In this case, Jess is suitable to develop rule-based systems where rules represent the heuristic knowledge of human experience in specific domains, and the knowledge base represents the state of an evolving situation.

However, Jess is general-purpose programming language exploitable even in case no rules have to be written. Particularly, Jess can directly access all Java classes and libraries, and is frequently used as a dynamic scripting or rapid application development environment. Whereas Java code generally must be compiled before it can be run, Jess interprets code and executes it immediately upon being typed. This allows us to experiment with Java APIs interactively and build up large programs incrementally. It is also relatively easy to extend the Jess language with new commands written in Java or in Jess itself, so the Jess language can be customized for specific applications.

Jess can be used in command-line applications, GUI applications, servlets, and applets. It is possible to develop Jess applications, with or without GUIs, without compiling a single line of Java code, but it is also possible to write Jess applications controlled entirely by Java code, with a minimum of Jess language code.

We observed that the most important step in developing a Jess application is to choose the proper architecture between the almost unlimited set of possibilities. One way to organize the possibilities is to list them in increasing order of the amount of Java programming involved:

1. Pure Jess language, with no Java code.

2. Pure Jess language, but the program accesses Java APIs.

3. Mostly Jess language code, but with some custom Java code in the form of new Jess commands written in Java.

4. Half Jess language code, with a substantial amount of Java code providing customs commands and APIs. Jess provides the `main()` function.

5. Half Jess language code, with a substantial amount of Java code providing customs commands and APIs. The programmer writes the `main()` function.

6. Mostly Java code, which loads Jess language code at runtime.

7. All Java code, which manipulates Jess entirely through its Java API.

In this thesis we concentrate on possibilities 5 and 6, since we should manipulate both Java and Jess commands. This is because context-aware applications consist not only of rules, but they also involve parts that should be implemented in Java.

Java is sometimes considered slow. This is not always true, since modern Java virtual machines are extremely powerful and fast. Being written in Java is not a

liability for Jess. The algorithm used for pattern matching is fairly efficient, since it allows Jess to search through large amounts of rules and facts in little time. Independent benchmarks have shown that Jess is significantly faster than many rule engines written in the C language, which is normally considered as faster than Java. For example, on many applications, Jess outperforms CLIPS by a factor of 20 or more on the same hardware.

Jess's rule engine uses an improved form of the Rete algorithm to match rules against the working memory. This algorithm explicitly trades space for speed. Jess does not contain commands that let us sacrifice some performance to decrease memory usage. Nevertheless, Jess's memory usage is reasonable.

Since Jess is a memory-intensive application, its performance is sensitive to the behavior of the Java garbage collector, which is the part of the Java Virtual Machine responsible for finding and deleting unused objects.

6.3 The Jess Language

The aim of this work is to use the functionality of the Jess rule engine in order to build a rule-based system suitable for context-aware applications.

We do not discuss the Jess language extensively in this section, but we rather provide the basic knowledge necessary to understand the case studies that we present in section 6.7. For the complete documentation we refer to [17] [18].

6.3.1 Basics

Symbols, numbers, strings, comments, lists, variables, and functions are the basic elements of the Jess language. Symbols are like identifiers in other languages, and in Jess they are case sensitive. The recommended symbols consist of letters, numbers, underscores, and hyphens; hyphens are traditional word separators. Jess gives special meaning to a few symbols that are like Java keywords. The symbol `nil` is like `null` in Java, and `TRUE` and `FALSE` are Jess's Boolean values. Other symbols have special meanings only in certain context, like the symbol `crlf` that is translated into a newline when printed.

The Jess language has three numeric types: `RU.INTEGER` corresponding to Java `int`, `RU.FLOAT` corresponding to Java `double`, and `RU.LONG` corresponding to Java `long`.

Character strings in Jess are delimited using double quotes ("). Strings are represented as `jess.Value` objects of type `RU.STRING`. All Jess values are represented internally by instances of the `jess.Value` Java class. This is the class used to interface Jess and Java code.

Comments add descriptive explanations to the Jess code that can appear anywhere in a program, since they are simply ignored for execution. Jess supports two kinds of programmer's comments: line comments and block comments. Line comments begin with a semicolon (;) and extend to the end of the line of text. Block comments work as in Java: they are delimited by the strings "/*" and "*/", at the beginning and the end, respectively.

Lists are the basic unit of structure in the Jess code. A list always consists of an enclosing set of parentheses and zero or more symbols, numbers, strings, or other lists. Lists are like arrays in Java and other languages. The first element of a list in Jess is called the list's *head*.

Variables are named containers that can hold a single value. They are like variables in Java. The main difference is that the Jess variables are *untyped*. This means that a Jess variable can hold a single value at a time (it can be of any type, like, for example, symbol, number, string or list), but this value and its type, can be changed during the lifetime of the variable. The variables in Jess are written as symbols beginning with a question mark (?) that is part of the variable's name. We do not need to declare variables in the Jess language, since they come into being when we first give them a value. To assign a value to a variable, the Jess language provides the `bind` function:

```
(bind ?x "The value")
```

For defining functions, Jess provides the `deffunction` construct, which has the form:

```
(deffunction <name> (<parameter>*)[<comment>] <expression>*)
```

The name of the `deffunction` must be a symbol. A function can have several parameters, where each of them must be a variable name complete with the question mark. The comment is optional and is a double quote string describing the purpose of the function. Finally, the body of the function can be composed of any number of expressions. Once a function is defined, it can be used like any other Jess function.

6.3.2 Scripting Java with Jess

We can create and manipulate Java objects directly from Jess. In this way, we can do virtually anything we can do from the Java code, except for defining new classes.

With the Jess `new` function is possible to create instances of Java classes. For example, consider we have a class named `Notification`. Suppose we want to call a method of this class within a rule written in Jess. For that, we need to create a Java object `Notification` and then store it in a variable with the following function call:

```
(bind ?class (new Notification))
```

In this way we obtain a reference to a Java object instance of the `Notification` class in the Jess variable `?class`, and we can invoke any of this object's methods using the `call` function. For example, we can invoke the method:

```
Public void SendNotification(String supervisor)
```

of the class `Notification`, using the following construct:

```
(call ?class SendNotification ?name)
```

where the variable `?name` is the argument of the method `SendNotification`, and it can be defined with the `bind` function as in:

```
(bind ?name "Patricia")
```

Like any Java code, Jess can only invoke the public constructors of public classes in other packages. Therefore, if we want Jess to be able to construct instances of the classes that we define, we must make sure that both the class and its constructors are defined as public.

When we call a Java method, Jess converts the arguments from Jess data types to Java types according to Table 3.

Table 3 - Standard conversion from Jess types to Java types

Jess type	Possible Java type
`RU.JAVA_OBJECT`	The wrapped object
The symbol `nil`	A null reference
The symbols `TRUE` and `FALSE`	`String`, `boolean` or `java.lang.Boolean`
`RU.SYMBOL`, `RU.STRING`	`String`, `char` or `java.lang.Character`
`RU.FLOAT`	`Float`, `double` or their wrappers
`RU.INTEGER`	`Long`, `short`, `int`, `byte`, `char`, and their wrappers
`RU.LONG`	`Long`, `short`, `int`, `byte`, `char`, and their wrappers
`RU.LIST`	A Java array

Likewise, Jess converts the return values of Java methods to Jess types according to Table 4. These conversions are generally the reverse of those in Table 3.

Table 4 - Standard conversion from Java types to Jess types

Java type	Jess type
A null reference	The symbol `nil`
A void return value	The symbol `nil`
`String`	`RU.STRING`
An array	A Jess list
`boolean` or `java.lang.Boolean`	The symbols `TRUE` and `FALSE`
`byte`, `short`, `int` or their wrappers	`RU.INTEGER`
`long` or `java.lang.Long`	`RU.LONG`
`double`, `float` or their wrappers	`RU.FLOAT`
`char` or `java.lang.Character`	`RU.SYMBOL`
anything else	`RU.JAVA_OBJECT`

6.4 Representing Facts in Jess

In this section we discuss how to deal with facts in Jess.

6.4.1 Basic commands by Jess prompt

This section presents some commands for creating and managing facts within the Jess working memory that we have used quite often in our work.

The `reset` command initializes the working memory and creates the fact `(MAIN::initial-fact)`. This fact has a special meaning for the Jess inner mechanism. Actually, many rules expect this initial fact to be in the working memory, since they cannot work correctly without it.

The `reset` command must be used at least once before using the working memory. For example, it is a good practice to use it when a program starts up, or at the beginning of an interactive session. Afterwards, we can issue `reset` again, whenever it is necessary to reinitialize the working memory.

The `watch` command prints messages whenever interesting changes occur in the working memory. We can pass different arguments to the `watch` function in order to allow Jess to report on several kinds of events. For example, the expression `watch facts` displays messages whenever any fact is added or removed into/from the working memory. The following code illustrates the use of the `reset` and `watch facts` commands:

```
Jess>(watch facts)
TRUE
Jess>(reset)
==> f-0 (MAIN:: initial-fact)
TRUE
```

In the responses Jess gives to the commands on the Jess prompt the ==> symbol means that a fact has been added to the working memory, while the <== symbol means that a fact has been removed from the working memory.

While the `watch facts` construct shows when new facts appear and old ones are removed, the `facts` command allow us to see the list of all the facts contained in the working memory. Continuing the previous example by typing `facts`, we can see the initial fact, which is the only one currently presents in the working memory:

```
Jess>(facts)
f-0 (MAIN:: initial-fact)
For a total of 1 facts in module MAIN
```

Jess is a rule engine for processing rules, and rules can only act on information represented by facts in the working memory. Therefore, it is necessary to know how to act on facts.

For adding a new fact in the working memory, Jess uses the `assert` function. The syntax of this function depends on the type of fact that we want to assert. We present these different types in section 6.4.2.

Each fact asserted in the working memory holds an index, called fact-id, which serves as a convenient way to refer to a fact when it is necessary to modify or retract it from the working memory, and when Jess decides the order in which rules have to be fired.

A fact represents a true statement about the world. Therefore, if changes occur in the world, it may be necessary to remove a fact from the working memory. For doing this, Jess offers the `retract` function. There are two ways to retract a fact: (i) by using the fact-id, as below:

```
Jess>(facts)
f-0 (MAIN:: initial-fact)
For a total of 1 facts in module MAIN
Jess>(retract 0)
TRUE
Jess>(facts)
For a total of 0 facts in module MAIN
```

(ii) by retracting the same fact using a reference to a `Fact` object, as in:

```
Jess>(facts)
f-0 (MAIN:: initial-fact)
For a total of 1 facts in module MAIN
Jess>(bind ?f (fact-id 0))
<Fact-0>
Jess>(retract ?f)
TRUE
Jess>(facts)
For a total of 0 facts in module MAIN
```

The latter way is faster, but it may be more convenient to use fact-ids if we are working interactively at the `Jess>` prompt.

6.4.2 Types of facts

This section presents the different types of facts supported by Jess. They are called *unordered*, *ordered* and *shadow facts*.

- An *unordered fact* is like a row in a relational database. For example:

```
(person (name "John Brown") (age 20) (gender Male))
```

This fact holds a head (`person`), and three properties, called *slots*, which are (`name`, `age`, `gender`). The head is like the table name in a database, and the slot names are like the column names. Slots may have single or multiple values. In the example above we have slots with single values, but we can create slots with multiple values by using the `multislot` keyword.

Before asserting an unordered fact with a given head, we have to specify its structure using the `deftemplate` construct. In section 6.6 we show how to use this command to define fact structures using the Java language.

- *Ordered facts* are Jess lists, where the *head* of the list defines a category for the fact. For example:

```
(shopping-list eggs milk bread)
```

Ordered facts are similar to unordered facts, the difference is that we can assert ordered facts without explicitly defining a `deftemplate`. In this case, Jess automatically generates an implied `deftemplate` for them that contains a single slot named `__data`:

```
(shopping-list (__data eggs milk bread))
```

Jess treats ordered facts in a special way, since it normally hides the name of the slot when the fact is displayed. This is just a syntactic shorthand, since sometimes slot names are redundant and force us to do more typing than we would like. Actually, ordered facts are unordered facts with a single multislot named `__data`.

- *Shadow facts* are the third category of facts supported by Jess. They are interesting since they connect the rule engine to the changes occurring in the real world outside the engine. Jess uses JavaBeans to implement shadow facts. A JavaBean consists of a self-contained, reusable software component that can be easily developed and assembled to create sophisticated applications. JavaBeans are special Java objects that contain values, called *properties*, which can change over time.

A shadow fact is an unordered fact whose slots correspond to the properties of a JavaBean. The name shadow fact derives from the remark that they are like images, or shadows, of JavaBeans outside Jess. Actually, shadow facts serve as the connection between the Jess working memory and the Java application inside which Jess is running.

In order to generate a template that represents a specific JavaBean for a shadow fact, we have to use the `defclass` function, instead of the `deftemplate`.

Moreover, we use the `definstance` construct, instead of the `assert` function, to add to the working memory an individual fact that is connected to one particular JavaBean instance.

6.5 Writing Rules in Jess

In this section, we discuss how to develop a knowledge base consisting of a collection of rules, which can take actions based on the current contents of the fact base.

6.5.1 Defining rules

The Jess pattern matcher compares the rules in the rule base to the facts in the working memory in order to decide which rule(s) to activate (see section 6.1). The list of the activated rules is contained in the agenda. If we wish to see if a rule is placed on (or removed from) this list, we can pass the argument `activations` to the `watch` function discussed in section 6.4.1. Actually, the command `watch activations` tells Jess to print a message whenever a rule is activated.

According to the conflict strategy adopted by Jess for ordering the activated rules, the first rule in the agenda is executed when we issue the `run` command. No rule is fired except during a call to `run`. Using the `run` command, the first rule activated in the agenda is executed. The rules may not only react to the contents of the working memory, but they may also change it. Therefore, one rule can add information into the working memory, which in turn can cause another rule to fire. The `run` function returns when there are no more rules to fire. Once a rule has fired, it does not fire again for the same list of facts. This means that the RHS of a certain rule already fired, cannot be executed twice if the facts contained in its LHS are not asserted again.

If we pass the argument `rules` to the `watch` function, Jess prints a message whenever a rule is fired. Finally, the `watch all` command tells Jess to print diagnostics for any important event that happens while the program is running. For example, we can see a message whenever a rule is activated and fired.

For defining a rule, Jess offers the `defrule` construct. Its general syntax is the following:

```
(defrule Name-of-the-rule “Comment”
(condition1)...(conditionN)=> (action1)...(actionM))
```

The name of the rule is a symbol where a hyphen (`-`) is often used to separate the words (this is a common practice, but it is not mandatory). The name is followed by an optional documentation string that describes the purpose of the rule. The symbol => separates the rule’s LHS (“if” part) from its RHS (“then” part), so the symbol => can be read as *then*.

We illustrate the Jess construct `defrule` for defining rules, and the inner mechanism of the Jess rule engine with an example of a simple rule.

6.5.2 Example of ECA rule in Jess

Consider the following ECA rule:

If ((John is the father of Bob) and (Bob is the father of Tom))
then (FindAndPrint the grandfather and the granchild)

We can express the LHS of the rule using two ordered facts for which we do not need to define a `deftemplate`, since Jess generates an implied one:

```
(Father John Bob)
(Father Bob Tom)
```

If John is the father of Bob and Bob is the father of Tom, this means that John is the grandfather of Tom. Therefore, we can define the rule:

```
(defrule FindAndPrintGrandfather
(Father ?x ?y) (Father ?y ?z) =>
(assert (Grandfather ?x ?z))
(printout t ?x " is the grandfather of " ?z crlf)
```

This rule assigns `?x` = John, `?y` = Bob, `?z` = Tom and, as soon we enter the `run` command, the RHS of the rule is executed. As a consequence, we could see that that the fact `(Grandfather John Tom)` is asserted in the working memory, and the message `"John is the grandfather of Tom"` is printed as output in the console. On the Jess console this example looks like:

```
Jess>(assert (Father John Bob))(assert (Father Bob Tom))
==> f-1 (MAIN:: Father John Bob)
<Fact-1>
Jess> ==> f-2 (MAIN:: Father Bob Tom)
<Fact-2>
Jess> (facts)
f-0 (MAIN:: initial-fact)
f-1 (MAIN:: Father John Bob)
f-2 (MAIN:: Father Bob Tom)
For a total of 3 facts in module MAIN
```

The `assert` function is used to add the two ordered facts`(Father John Bob)` and `(Father Bob Tom)`. The `facts` command shows all the facts currently contained in the working memory.

When we insert the `FindAndPrintGrandfather` rule previously defined, Jess prints the following text:

```
==> Activation: MAIN: FindAndPrintGrandfather: f-1, f-2
MAIN :: FindAndPrintGrandfather: +1+1+1+2+t
```

This text indicates that the rule matches the facts `f-1` and `f-2` and is activated in the agenda. The line `MAIN :: FindAndPrintGrandfather: +1+1+1+2+t` is a message printed by the `watch all` command, which shows how Jess interprets the rules internally.

Finally, when we enter the `run` command, the activated rule is fired:

```
Jess> (run)
FIRE 1 MAIN :: FindAndPrintGrandfather f-1, f-2
==> f-3 (MAIN :: Grandfather John Tom)
John is the grandfather of Tom
```

The RHS of the rule is executed, so that the new fact `(Grandfather John Tom)` is added to the working memory, and the output `John is the grandfather of Tom` is displayed.

6.5.3 Qualifying patterns with conditional elements

For writing rules with more complex relationships between facts, we need to know how to qualify patterns with conditional elements. Patterns are individual facts that must be true for executing the rule. The LHS of a rule can consist of a list of zero or more patterns. With the conditional elements (CE), we can group patterns into logical structures.

- *The "and" conditional element*. Any number of patterns can be enclosed in a list with `and` as the head. The resulting pattern is matched if and only if all of the enclosed patterns are matched. An example of `and` conditional element in the LHS of a rule is:

```
(and (> ?x 30) (< ?x 50) (> ?y 20) (< ?y 60))
```

The RHS of this rule is executed if and only if all the patterns match, namely if 30 < `?x` < 50 and, at the same time, 20 < `?y` < 60. The same result would be obtained if the `and` conditional element was omitted. The `and` conditional element is interesting when combined with `or` and `not` conditional elements, since it can be used for constructing more complex and interesting logical conditions.

- *The "or" conditional element.* Any number of patterns can be enclosed in a list with `or` as the head. The resulting pattern is matched if at least one of the enclosed patterns matches. An example is:

```
(or (> ?x 30) (< ?x 50) (> ?y 20) (< ?y 60))
```

This expression means that the RHS of this rule is executed if one or more of the patterns match.

The `or` conditional element can be used inside an `and` group and vice versa. An example of `or` inside an `and` conditional element is:

```
(and (or (> ?x 30) (< ?x 50) (> ?y 20) (< ?y 60))
     (neq ?x 0)
     (neq ?y 0))
```

The RHS of this rule is executed if at least one of the patterns inside the `or` match and, at the same time, `?x` and `?y` are not equal to zero. Jess automatically rearranges the patterns inside the `or` CE as follows:

```
(or  (and (> ?x 30)(neq ?x 0) (neq ?y 0)
     (and (< ?x 50)(neq ?x 0) (neq ?y 0)
     (and (> ?y 20)(neq ?x 0) (neq ?y 0)
     (and (< ?y 60)(neq ?x 0) (neq ?y 0))
```

Consequently, a rule containing an `or` conditional element with n branches (in our case n = 4) is equivalent to n rules, each of which has one of the branches on its LHS. Each of these generated rules is called a *subrule*. Jess knows that the subrules created from a given rule are related. Therefore, if the original rule is removed, every subrule associated with that rule is also removed.

➢ *The "not" conditional element*. Most patterns can be enclosed in a list with `not` as the head. The resulting pattern is matched if a fact (or a set of facts) that matches the pattern inside the `not` is not found. For example:

```
(not (> ?x 30))
```

The RHS of this rule is executed for any value of `?x` that is not bigger than 30, namely for every value of `?x` that is lower or equal to 30.

The `not` CE is different from the other conditional elements, since a `not` pattern matches the *absence* of a fact. Actually, if we want to define that the value of `?x` is not contained between 30 and 50, we cannot state:

```
(and (not (> ?x 30) (< ?x 50))
```

This line of code generates the following Jess exception:

```
Jess reported an error in routine Group.add
Message: CE is a unary modifier not.
```

The correct syntax is therefore:

```
(and (not(> ?x 30)) (not(< ?x 50)))
```

The `not` CE can be used in arbitrary combinations with the `and` and `or` conditional elements. For example, consider the LHS defined before:

```
(and (or (> ?x 30) (< ?x 50) (> ?y 20) (< ?y 60))
     (neq ?x 0)
     (neq ?y 0))
```

This LHS can be written using the `not` conditional element as:

```
(and (or   (not(< ?x 30))
           (not(> ?x 50))
           (not(< ?y 20))
           (not(> ?y 60)))
     (neq ?x 0)
     (neq ?y 0))
```

➢ *The "test" conditional element*. We can have a pattern enclosed in a list with `test` as the head. This case is special, since the body does not consist of a pattern to match against the working memory, but of a Boolean function. A `test` pattern fails if and only if the function evaluates to the symbol `FALSE`. If it evaluates to `TRUE`, the pattern succeeds. For example, if we want to evaluate if the value of ?x is contained between 30 and 50 and the value of `?y` between 20 and 60, we can use the following LHS:

```
(test (and (> ?x 30) (< ?x 50) (> ?y 20) (< ?y 60)))
```

This expression means that if all the patterns with `and` as the head match, then the RHS of the rule is executed, since the body of the `test` CE is evaluated `TRUE`.

Finally, the `test` and `not` conditional elements can be combined. For example, `(not (test (eq ?x 0)))` is equivalent to `(test (neq ?x 0))`.

6.6 Embedding Jess in Java Applications

In this chapter we have dealt so far with small example applications that use the command-line interface tool `jess.Main`. However, in order to deploy larger applications, like, e.g., a web server, an application server or a browser, we have to use the *Jess library*. In fact, `jess.Main` is just a command-line wrapper built around the Jess library.

With the Jess library, we can create any number of individual Jess inference engines, define rules for them, add data to their working memories, run them in different threads and collect the generated results. All these objectives can be reached from Java code, without using `jess.Main`. Below we explain how this is possible.

6.6.1 The jess.Rete class

The core of the Jess library is the `jess.Rete` class. We can think of an instance of this class as a sort of instance of Jess. Different `Rete` objects are engines completely independent of each other, since each `jess.Rete` object has its own independent working memory, list of rules and set of functions to manipulate. The `Rete` class exports methods for adding, finding and removing facts, rules, functions and other constructs. The `Rete` class constitutes a convenient central access point for embedding Jess in Java applications.

To create a `jess.Rete` object, we can use the default constructor without arguments:

```
import jess.*;
     ...
     Rete engine = new Rete();
```

6.6.2 The executeCommand method

Once we have created a `Rete` object, the simplest and most powerful way to manipulate Jess from Java consists of using the `executeCommand` method. This method accepts a `String` argument and returns a `jess.Value`. The `String` is interpreted as an expression in the Jess language, and the return value is the result of evaluating the expression. For example, if we want to assert the fact `(Father John Bob)`, we can do the following:

```
import jess.*;
     ...
     Rete engine = new Rete();
     engine.executeCommand("(assert(Father John Bob))");
```

If we want to run the engine, we can use the command:

```
     engine.executeCommand("(run)");
```

The `executeCommand` is convenient, but it has some important drawbacks. It is *verbose*, since besides typing the whole Jess command, we have also to type the `executeCommand`. Moreover it is *potentially inefficient*, since Jess has to parse the arguments before executing the command. If the Jess code passed as argument is long enough to justify the use of the `executeCommand` (for instance, a Jess

rule), the problem does not exist. However, a single short Jess command as argument, like `reset` or `run`, could cause a big overhead. Finally, it is *error-prone*, since the Java compiler cannot detect syntax errors in the Jess script so that errors cannot be found until the program is executed.

For all these reasons, whenever possible we should consider using Jess's direct Java APIs. Some `Rete` methods that correspond to simple Jess functions are shown in Table 5.

Table 5 - Examples of simple methods in the `jess.Rete` class

Rete method	Jess equivalent
`reset()`	`(reset)`
`run()`	`(run)`
`clear()`	`(clear)`

Therefore, if we want to initialize the engine and run it, we just need to write the code:

```
engine.reset();
...
engine.run();
```

6.6.3 Working with Fact objects in Java

When we are working with facts in Java, it is better to directly construct `jess.Fact` objects rather than use the `executeCommand`. The `jess.Fact` class is a subclass of `ValueVector`. The `jess.ValueVector` class is the Jess's internal representation of a list, therefore it has a central role in the Jess programming in Java. All the entries in the `ValueVector` correspond to slots of the `Fact`. Table 6 lists some of the Java methods helpful for working with `Fact` objects.

Table 6 - Examples of `jess.Rete` methods for working with facts

Rete method	Jess equivalent
`addDeftemplate(Deftemplate)`	`(deftemplate)`
`assertFact(Fact)`	`(assert fact)`
`findfactByFact(Fact)`	None
`findfactById(int)`	`(fact-id number)`
`retract(Fact)`	`(retract fact-id)`

Every `jess.Fact` has an associated `jess.Deftemplate` object for describing the slots that the fact can have. The `Deftemplate` definition includes a list of

slots, each of which can have a default value and a type. The `addDeftemplate(Deftemplate)`construct in Java serves to add `Deftemplate` objects to the `Rete` class, and it is equivalent to the Jess `deftemplate` command.

To build a useful `Deftemplate`, we need to specify the name to the constructor, and then add the named slots one by one. If we have defined a `Deftemplate` with a given head for a fact, we do not need to define it again, since all `Fact`s with the same head can share this `Deftemplate`. We show how to create `jess.Deftemplate` objects in section 6.7.

The `Rete` method `assertFact(Fact)` can be used to assert a `Fact` object. This method is more convenient than using the `executeCommand("(assert fact)")`. Once we assert a `Fact` object, it becomes part of the `Rete` object's internal data structures and we cannot change the values of any of its slots. In contrast, the Jess prompt allows slot values to be changed using the `modify` function, but from Java we have to retract the fact with the `retract` method in order to release the `Fact` object and alter it as we wish. The `retract` method accepts a `Fact` object as argument. If we asserted a fact from Java, we have a reference to the corresponding `Fact` object. When we want to retract this fact, we just need to pass the object as argument to `retract`. For example:

```
Fact f = ...
engine.retract(f);
```

If we asserted a fact from Jess, we do not have a reference to a corresponding `Fact` object. In order to retract the fact we have two options:

- We can construct a `Fact` object in Java just like the one we want to retract. Then, we can pass this `Fact` object as argument to `retract`. This approach is convenient, since the working memory is stored as a hash table with the `Fact` objects as the key. Therefore, a `Fact` object can be quickly found in the working memory.

- If we do not want to construct the duplicate of the Jess fact to retract from Java, we can use the Java `findfactById(int)` method that finds a fact given its numeric identifier. This implies that we should know the numeric identifier of the Jess fact. Then, we can pass the found `Fact` object to `retract`. This approach is slow, since to find out the fact using a particular numeric identifier requires processing time. Actually, Jess has to examine each `Fact` in the working memory until it finds the intended one.

Finally, the `jess.Rete` class holds a method that does not have a correspondence in the Jess prompt: the `findfactByFact(Fact)`method. It is useful when we have already defined a `Fact` object in Java and we want to get a reference to it.

6.6.4 Working with rules in Java

Defining rules from Java is a complex and badly documented process. Therefore, as it is shown in the following sections, our approach has been to create rules exploiting the expressiveness of the Jess language.

6.6.5 *ECA rule's example*

Consider the same ECA rule presented as example in section 6.5.2:

If ((John is the father of Bob) and (Bob is the father of Tom))
then (FindAndPrint the grandfather and the granchild)

We want to express the LHS of the rule using the two ordered facts `(Father John Bob)` and `(Father Bob Tom)`. We do not need to define a `Deftemplate` for them, since Jess automatically generates an implied template for ordered facts. We just need to create a `jess.Rete` class and to construct two `Fact` objects as follow:

```
import jess.*;
...
Rete engine = new Rete();
engine.reset();

Fact f1 = new Fact("Father",engine);
f1.setSlotValue("__data", new Value(new ValueVector().
          add(new Value ("John", RU.SYMBOL)).
          add(new Value ("Bob", RU.SYMBOL)), RU.LIST));
engine.assertFact(f1);

Fact f2 = new Fact("Father", engine);
f2.setSlotValue("__data", new Value(new ValueVector().
          add(new Value ("Bob", RU.SYMBOL)).
          add(new Value ("Tom", RU.SYMBOL)), RU.LIST));
engine.assertFact(f2);
```

In the listing above we specified a value for each slot in each `Fact`. This is necessary, otherwise Jess would use default values when the fact is asserted.

The aim of the rule is to find from the facts `(Father John Bob)` and `(Father Bob Tom)` who is the grandfather, and to print a message that indicates that. If John is the father of Bob and Bob is the father of Tom, this means that John is the grandfather of Tom. Therefore, we can define the rule:

```
engine.executeCommand("(bind ?class (new RightPortion))" +
          "(defrule FindAndPrintGrandfather" +
          "(Father ?x ?y)(Father ?y ?z)=>" +
          "(assert (Grandfather ?x ?z))" +
          "(call ?class PrintGrandfather ?x ?z))");
engine.run();
```

In the LHS of the rule, the command `(bind ?class (new RightPortion))` creates an object instance of the `RightPortion` class and stores this object in variable `?class`. The `RightPortion` class contains a `PrintGrandfather` method for printing a message as output. Moreover, the LHS of the rule assigns `?x` = John, `?y` = Bob, `?z` = Tom. With the command `run()` the engine executes the RHS. Therefore, the fact `(Grandfather John Tom)` is asserted and the command `(call ?class PrintGrandfather ?x ?z)` calls the `public void PrintGrandfather(String x, String z)` method defined in the `RightPortion` class.

If we also want to turn on all the `watch` diagnostic from Java code (the equivalent of `watch all` in Jess), we can do the following:

```
Context context = engine.getGlobalContext();
Funcall fc = new Funcall("watch",engine);
fc.arg("all");
fc.execute(context);
```

The result of this program is exactly the same as the one that we have obtained in section 6.5.2 by using the `Jess>` prompt. The only difference is that, in this case, the application is controlled by Java code.

```
 ==> Focus MAIN
 ==> f-0 (MAIN::initial-fact)
 ==> f-1 (MAIN::Father John Bob)
 ==> f-2 (MAIN::Father Bob Tom)
 ==> Activation: MAIN::FindAndPrintGrandfather :  f-1, f-2
 MAIN::FindAndPrintGrandfather: +1+1+1+2+t
 FIRE 1 MAIN::FindAndPrintGrandfather f-1, f-2
 ==> f-3 (MAIN::Grandfather John Tom)
 John is the grandfather of Tom
 <== Focus MAIN
```

6.7 Jess Case Studies

For concluding this chapter, we present some Jess case studies, which are the result of the implementation work of this thesis. They are simple applications that show how to use the Jess rule engine in a context-aware scenario. Although they are simple, these examples are a good starting point for implementing large and complex rule-based systems, since they show the usage of most of the basic concepts we have presented in this chapter. Particularly, they illustrate how to use the Jess's Java APIs, by exploiting the expressiveness of the Jess language to write ECA rules.

Consider the context-aware scenario where the supervisor of Laura wants to receive a notification message if Laura is working. We can represent this circumstance in the real world with the following ECA rule in our application:

If (Laura is working) *then* (Send notification to her supervisor).

The first step consists of defining the LHS of the rule using the ordered fact `State` with the value `(Laura-is-working)`:

```
Fact f = new Fact("State", engine);
f.setSlotValue("__data", new Value(new ValueVector().
add(new Value("Laura-is-working",RU.SYMBOL)), RU.LIST));
```

We define a `public void SendNotification(String supervisor)`method in the class `Notification`, which accepts a `String` as argument (the name of the Laura's supervisor) and displays a message to notify that Laura is working. For calling this method within the `executeCommand`, we need to create a `Notification` object in Jess, and then to store it in a variable (`?class`). In this way, we have a reference to the Java object `Notification` in the Jess variable `?class`, and we can invoke its `SendNotification` method using the `call` function:

```
engine.executeCommand("(defrule LauraRule (State ?x) => " +
                "(bind ?class (new Notification))" +
                "(bind ?name \"Patricia\")" +
                "(call ?class SendNotification ?name))");
```

The variable `Patricia` is the argument that we pass to the method `SendNotification` and it is defined using the `bind` function.

Once the fact and the rule are defined, the user may choose different options that allow him: (i) to add the fact to the working memory and see the rule being fired; (ii) to retract the fact if it is already in the working memory; and (iii) to exit from the application. Particularly:

1. The first option asserts the fact (`State Laura-is-working`), and add it in the working memory. Then it runs the engine and fires the rule as follows:

   ```
   engine.assertFact(f);
   ...
   engine.run();
   ```

   The output printed in this case is like:

   ```
   ==> f-1 (MAIN::State Laura-is-working)
   ==> Activation: MAIN::LauraRule :  f-1
   FIRE 1 MAIN::LauraRule f-1

   Patricia, Laura is working
   ```

2. The second option initially checks if the fact (`Laura-is-working`) is in the working memory. If it is, the fact is retracted from the working memory and the message "`Patricia, Laura is not working`" is sent to the Laura's supervisor. If the fact is not in the working memory an error message is displayed.

3. Finally, the third option allow the user to exit from the application and the engine is reinitialized with the command `engine.reset()`.

Consider a variation of the scenario just described. This time we want that the application notifies Patricia whenever Laura is in the laboratory. If Laura's location changes, particularly when Laura leaves the laboratory, her supervisor is interested to know it.

We represent the location of Laura as a point in the space identified by two Cartesian coordinates (x, y). We represent the location of the laboratory as a rectangle identified by four Cartesian coordinates (x1, x2; y1, y2). While Laura's location changes over time, the laboratory's location is static, therefore we can use fixed values for representing this location.

We have described these circumstances with the following ECA rules:

1. *If* (Laura is in the Lab) *then* (Send to her supervisor the notification "Laura is working");

2. *If* (Laura is not in the Lab) *then* (Send to her supervisor the notification "Laura is not working").

We have defined the unordered facts:

1. `(Laura-location (slot coordx (type INTEGER))(slot coordy (type INTEGER)))`. Before asserting this fact we need to define a `Deftemplate`:

```
Deftemplate d1 = new Deftemplate("Laura-location",
"It shows in a static way the location of Laura ",
engine);
d1.addSlot("coordx", Funcall.NIL, "INTEGER");
d1.addSlot("coordy", Funcall.NIL, "INTEGER");
engine.addDeftemplate(d1);
```

2. `(Lab-location (multislot LabCoordinates))`. The `Deftemplate` for it is:

```
Deftemplate d2 = new Deftemplate("Lab-location", "It
defines the location of the lab using 4 points",
engine);
d2.addMultiSlot("LabCoordinates", Funcall.NILLIST);
engine.addDeftemplate(d2);
```

While the fact `Laura-location` is dynamic and needs to be asserted or retracted according to the changes in Laura's coordinates, the fact `Lab-location` is static. Therefore we can directly add it to the working memory as shown:

```
Fact f2 = new Fact("Lab-location", engine);
ValueVector LabLoc = new ValueVector();
LabLoc.add(new Value(coordx1, RU.INTEGER));
LabLoc.add(new Value(coordx2, RU.INTEGER));
LabLoc.add(new Value(coordy1, RU.INTEGER));
LabLoc.add(new Value(coordy2,RU.INTEGER));
f2.setSlotValue("LabCoordinates",new Value(LabLoc,RU.LIST));
engine.assertFact(f2);
```

The fact `Laura-location` is asserted if Laura's coordinates are contained between the values (x1, x2) and (y1, y2), i.e., whenever Laura is in the laboratory. For this purpose, we have created a simulation that generates fictitious locations for Laura. These values are assigned to the slots `coordx` and `coordy` of the `Fact f1 = new Fact("Laura-location", engine)`. The loop asserts this fact and runs the engine, therefore one of the following two rules is fired according to the values issued:

```
engine.executeCommand("(defrule rule1 " +
"?temp1 <- (Laura-location (coordx ?x)(coordy ?y))" +
"(Lab-location (LabCoordinates ?x1 ?x2 ?y1 ?y2))" +
"(test (and (> ?x ?x1)(< ?x ?x2)(> ?y ?y1)(< ?y ?y2)))" +
" => (bind ?class (new Notification))" +
"(bind ?name \"Patricia\")" +
"(call ?class SendNotification1 ?name)" +
"(retract ?temp1))");

engine.executeCommand("(defrule rule2 " +
"?temp2 <- (Laura-location (coordx ?x)(coordy?y))" +
"(Lab-location (LabCoordinates ?x1 ?x2 ?y1 ?y2))" +
"(test (and (or (< ?x ?x1)(> ?x ?x2)(< ?y ?y1)(> ?y ?y2))
           (neq ?x 0)(neq ?y 0)))" +
" => (bind ?class (new Notification))" +
```

```
"(bind ?name \"Patricia\")" +
"(call ?class SendNotification2 ?name)" +
"(retract ?temp2))");
```

Rule1 assigns the issued values to Laura-location and stores this fact in variable ?temp1. If all patterns inside the and conditional elements match, the expression with test as head evaluates as TRUE and the RHS of the rule invokes the method SendNotification1 of the class Notification . This method prints the message "Patricia, Laura is in the lab". Finally, the fact Laura-location stored in the variable ?temp1 is retracted from the working memory to release the values of the Laura's coordinates for a new cycle.

Likewise, rule2 assigns the issued values to Laura-location and stores this fact in variable?temp2. If at least one of the patterns inside the or conditional elements matches and, at the same time, both the variables ?x and ?y are not equal to zero, then the expression with test as head evaluates as TRUE and the RHS of the rule invokes the method SendNotification2 of the class Notification . This method prints the message "Patricia, Laura is not in the lab". Finally, the fact Laura-location stored in the variable ?temp2 is retracted from the working memory to release the values of the Laura's coordinates for a new cycle.

7 Mapping ECA-DL to Jess

This chapter discusses the mapping of ECA-DL rules onto Jess rules, and provides guidelines for realizing and generalizing this mapping.

This chapter is structured as follows: section 7.1 states the goal of the chapter and the approach that has been used to reach this goal. Section 7.2 presents examples of mapping from ECA-DL to Jess. Section 7.3 proposes guidelines for mapping from ECA-DL rules constructs to Jess counterparts. Finally, Section 7.4 presents some concluding remarks.

7.1 Goal and General Approach

The ultimate goal of our work has been to define a mapping from ECA-DL to a well-known rule-based engine, particularly to Jess. This is an important part of the design and the implementation of the Controller component (see Figure 3), since it allows ECA rules written in ECA-DL to be executed in a robust and powerful environment.

The aim of this chapter is to define how to use Jess for executing ECA-DL rules. Particularly, we have tried to answer the following specific questions:

- How to provide a general mapping from ECA-DL to the Jess engine?
- How to write ECA-DL rules in the specific Jess language?
- How to map constructs of the ECA-DL to constructs of Jess?

The order of these questions reflects the approach that we have used for designing the mapping. Our approach towards this mapping is depicted in Figure 28.

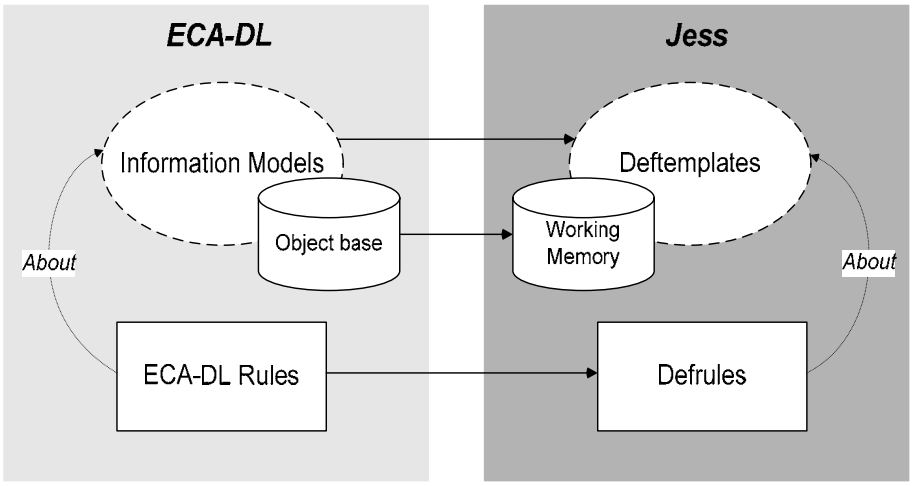

Figure 28 – Design of the mapping from ECA-DL onto Jess: general approach

An information model in ECA-DL consists of a static UML model that depicts entities and contexts, which reflect the knowledge that the target context-aware application manipulates. Entities and context are represented as classes, and the relationships between them are defined as associations between these classes. A `deftemplate` in Jess is the static structure to define the structure of facts. We need to define `deftemplates` before asserting any facts in the working memory of the rule engine. The first step in our approach has been to provide a mapping from the ECA-DL information models proposed in Chapter 3 (i.e., the UML models), to `deftemplates` in Jess.

We can create instances of the classes represented in an ECA-DL information model. These instances are the objects contained in the object base shown in Figure 28. Analogously to defining objects in ECA-DL, we can also assert facts with specific values in Jess. These facts reflect the structure of the `deftemplates` and they are contained in the working memory of the Jess engine. Therefore, the operation to create objects in ECA-DL corresponds to the operation to assert facts in Jess. The second step in our approach has been to provide a mapping from one or more ECA-DL objects in the object base to facts in the working memory of Jess.

Figure 28 shows that ECA-DL rules are based on information models. Analogously, Jess rules, defined with the `defrule` command, are based on `deftemplates`. Actually, ECA-DL rules use objects that are instances of entity and context classes of the corresponding information model. Likewise, `defrule` constructs in Jess uses facts asserted on previously defined `deftemplates`. The third and last step in our approach has been to provide a mapping from ECA-DL rules to `defrules` by investigating the correspondences between ECA-DL specific constructs to Jess constructs. In summary, our approach towards the mapping includes:

- Mapping of information models in ECA-DL to Jess `deftemplates`;
- Mapping of objects in the object base to facts in the Jess engine; and
- Mapping of ECA-DL rules to Jess rules.

7.2 Examples

We present here five examples of mapping from ECA-DL rules (see section 3.3) to Jess rules by using the approach mentioned above. By studying these examples, we have been able to identify patterns of mapping that can be generalized. In the following sections we discuss this generalization and we provide guidelines on how the mapping should proceed.

For each example we propose:

- The corresponding scenario in natural language and the corresponding ECA rule;
- The mapping of the ECA-DL information model to Jess `deftemplate`s;
- The mapping of the ECA-DL rule on a Jess rule.

7.2.1 Example 1

"Patricia would like to be notified when Laura enters the laboratory, without friends, and Laura's computer is on".

We have expressed this scenario by using the following ECA rule (see section 3.3.1):

If <Laura is in the lab AND (NOT Laura is with friends) AND Laura's computer is on> *then* <Notify (Patricia), "Laura is working.">

Figure 29 shows the general mapping from the ECA-DL information model to Jess `deftemplate`s.

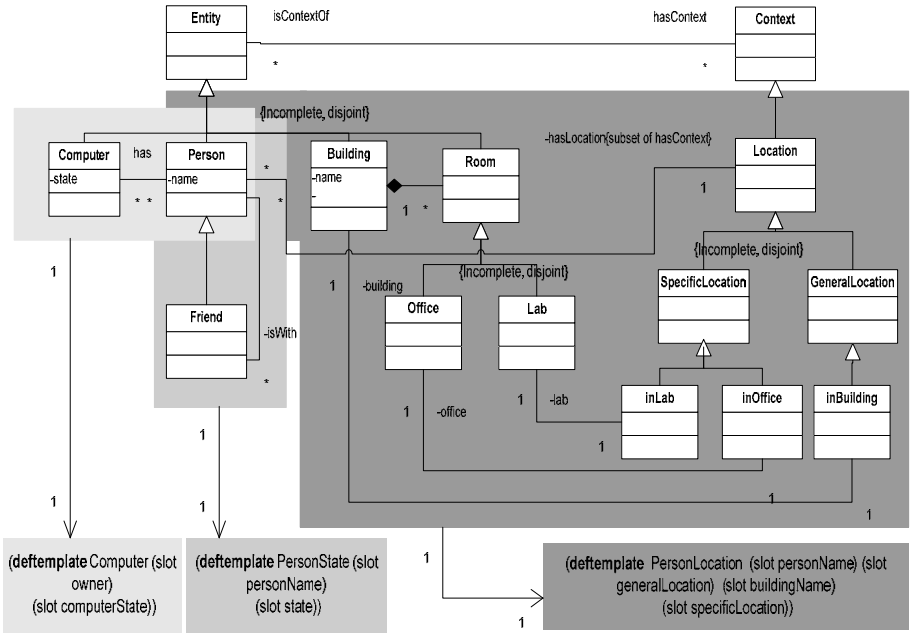

Figure 29 – General mapping of Example 1

Figure 29 depicts the association hasLocation between the entity Person and the context Location. This association has been mapped on a `deftemplate` called

PersonLocation. A Person has a name (slot personName in the template), and may have a GeneralLocation (slot generalLocation). This GeneraLocation may be InBuilding, in which case the entity Building has a name (slot buildingName). A person may also have a SpecificLocation (slot specificLocation in the template), which may be inLab or inOffice.

Figure 29 depicts the associations Person isWith Friend and Person has Computer. The association Person isWith Friend has been mapped on a `deftemplate` called PersonState. A Person (slot personName in the template) may be with Friend (slot state). The association Person has Computer has been mapped on a `deftemplate` called Computer. A Person (slot owner in the template) may have a Computer, and Computer has a state (slot computerState), which may be On or Off.

We have used the following ECA-DL rule in order to express the considered ECA rule (see section 3.3.1):

```
Upon EnterTrue (Laura.inLab)
When (NOT (Laura.isWithFriends)) AND (Laura.hasComputerOn)
Do Notify (Patricia, "Laura is working.")
Always
```

This ECA-DL rule has been mapped on the following Jess rule:

```
(defrule example1
(PersonLocation (personName Laura)(specificLocation inLab))
(PersonState (personName Laura)(state ~isWithFriends))
(Computer (owner Laura)(computerState on))
=>
(bind ?class (New Notification))
(bind ?name "Patricia")
(call ?class SendNotification ?name))
```

The **`defrule`** checks whether the Jess working memory has been asserted a fact `PersonLocation` that has a slot `personName` with value `Laura` and a slot `specificLocation` with value `inLab`. Moreover, it checks for a fact `PersonState` that has a slot `personName` with value `Laura` and a slot `state` that does not have value `isWithFriends`. Finally, it checks for a fact `Computer` that has a slot `owner` with value `Laura` and a slot `computerState` with value `On`.

If the engine finds all these facts, it executes the RHS of the rule, which creates an object named `?class` by instantiating the `Notification` class, assigns the string `"Patricia"` to a variable `?name`, and, finally, calls method `SendNotification` on the `?class` object. The parameter `?name` defines the name of the person to whom this notification should be send, in this case to Patricia.

The **`Upon`** and **`When`** clauses of the ECA-DL rule have been mapped onto the LHS of the **`defrule`**, while the **`Do`** clause has been mapped onto the RHS.

7.2.2 Example 2

"When Laura and Patricia start a meeting together, the meeting time should be counted".

We have expressed this scenario by using the following ECA rule (see section 3.3.2):

If <Laura is in meeting AND Patricia is in meeting AND Laura and Patricia share the meeting> *then* <Count meeting time>

Figure 30 shows the general mapping from the ECA-DL information model to Jess `deftemplate`s.

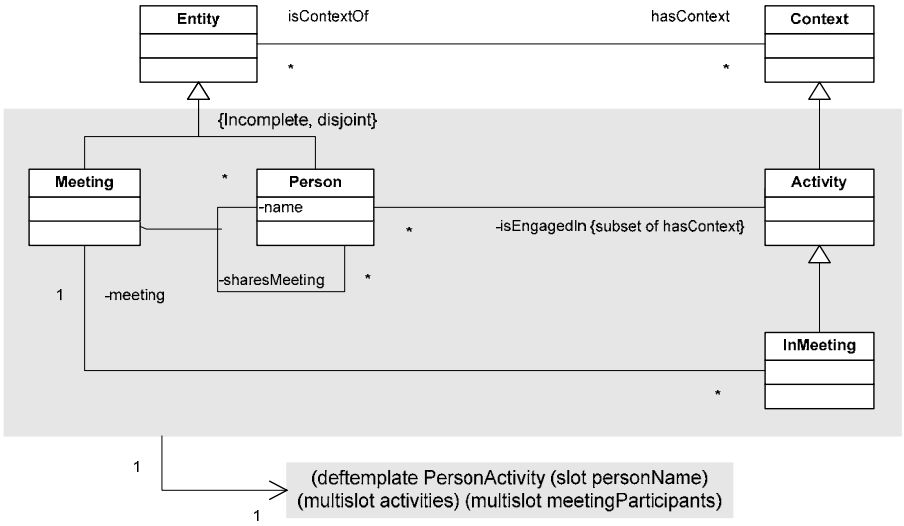

Figure 30 – General mapping of Example 2

Figure 30 depicts the association isEngagedIn between the entity Person and the context Activity. This association has been mapped on a `deftemplate` called PersonActivity. A Person has a name (slot personName in the template), and may be engaged in one or more Activitities (multislot activities). In case one of these activities is InMeeting, the person who is in the meeting may share this meeting with another person (or more persons). This explains why Meeting is defined in the information model as an association class relating two persons.We have mapped this association by adding to the same `deftemplate` PersonActivity with the multislot meetingParticipants.

We have used the two following ECA-DL rules in order to express the considered ECA rule (see section 3.3.2):

```
Upon EnterTrue (Laura.inMeeting)
     AND EnterTrue (Patricia.inMeeting)
When Laura.sharesMeeting (Patricia)
Do StartCountMeetingHours
Always

Upon TrueToFalse (Laura.inMeeting)
     OR TrueToFalse (Patricia.inMeeting)
When Laura.sharesMeeting (Patricia)
Do StopCountMeetingHours
Always
```

These ECA-DL rules have been mapped on the following Jess rules:

```
(defrule firstRuleExample2
(PersonActivity (personName Laura)(activities inMeeting)
```

```
(meetingParticipants Patricia))
(PersonActivity (personName Patricia)(activities inMeeting))
=>(call MeetingHours StartCountMeetingHours))


(defrule secondRuleExample2
(or  (PersonActivity (personName Laura)(activities ~inMeeting)
     (meetingParticipants Patricia))
     (PersonActivity (personName Patricia)
     (activities ~inMeeting))
)
=>(call MeetingHours StopCountMeetingHours))
```

The first **`defrule`** checks whether the Jess working memory has been asserted a fact `PersonActivity` that has a slot `personName` with value `Laura`, a slot `activities` with value `inMeeting`, and a slot `meetingParticipants` with value `Patricia`. Moreover, it checks for another fact `PersonActivity` that has a slot `personName` with value `Patricia` and a slot `activities` with value `inMeeting`.

If the engine finds both facts, it executes the RHS of the rule that calls the method `StartCountMeetingHours` on the `MeetingHours` class. `StartCountMeetingHours` is a static method for which we do not need to initialize the `MeetingHours` class. Since static methods do not support multiple instances, we can have only one instance of the `StartCountMeetingHours` method. Therefore, only one application at a time can use the `MeetingHours` class, i.e., if we have a rule that starts counting the time of the meeting between Laura and Patricia, it has to be followed by another rule that stops counting this time before another meeting time starts to be counted .

The second **`defrule`** checks whether the Jess working memory has been asserted at least one of the facts `PersonActivity` of the previous rule, with the only difference that in this case the value of slots `activities` must be `~inMeeting`, since when Laura or Patricia leave the meeting we execute the RHS of the rule by calling the method `StopCountMeetingHours` on the `MeetingHours` class previously instantiated.

The **`Upon`** and **`When`** clauses of the ECA-DL rules have been mapped onto the LHS of the **`defrule`**s, while the **`Do`** clauses have been mapped onto the RHS.

7.2.3 Example 3

"During the hot season, when the temperature in a building of the University of Twente is more than 30 degrees and it is later than 14:00 hours and earlier than 17:00 hours, all the persons in the building should be notified to go home".

We have expressed this scenario by using the following ECA rule (see section 3.3.3):

If <During the hot season the temperature in a building of the University of Twente is more than 30 degrees AND it is later than 14:00 hours AND it is earlier than 17:00 hours > *then* <Notify (all the persons in the building), "You can go home.">

Figure 31 shows the general mapping from the ECA-DL information model to Jess `deftemplates`.

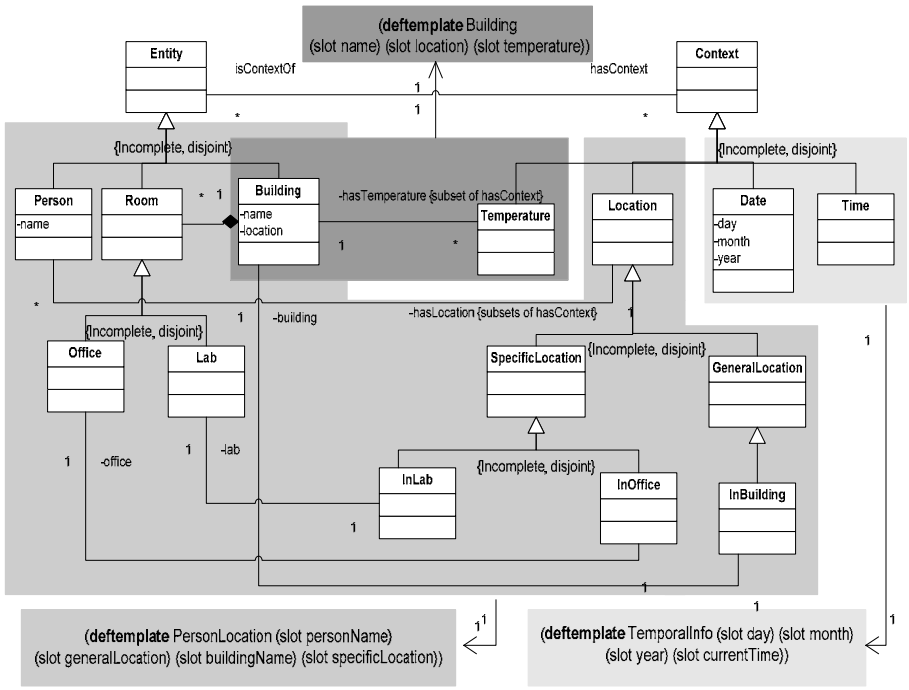

Figure 31 – General mapping of Example 3

Figure 31 depicts the association hasLocation between the entity Person and the context Location. This association has been mapped onto the PersonLocation `deftemplate` defined in Example 1.

Figure 31 also depicts the association hasTemperature between the entity Building and the context Temperature, and the context Date and Time. The association Building hasTemperature has been mapped onto a `deftemplate` called Building. A Building has a name (slot name in the template), a location (slot location), which in our case is the University of Twente, and a temperature (slot temperature). The context Date and Time have been mapped onto a `deftemplate` called TemporalInfo. Date has attributes day, month, year that are mapped, respectively, onto the slots day, month, year of the `deftemplate`, while Time has been mapped onto the slot currentTime.

We have used the following ECA-DL rule in order to express the ECA rule of this example:

```
Scope (Select (building.*, build, build.inUT); b))
{
Upon EnterTrue (b.temperature > 30)
When (currentTime > 14) AND (currentTime < 17)
Do Notify (Select (person.*, p, p.InBuilding(b)),"You can go
home.")
from <May> to <September>
}
```

This ECA-DL rule has been mapped on the following Jess rule:

```
(defrule example3
(PersonLocation (personName ?p)(generalLocation inBuilding)
```

```
(buildingName ?b1))
(Building (name ?b2&:(eq ?b2 ?b1))
(location inUT)(temperature ?temp&:(> ?temp 30)))
(TemporalInfo (month May|June|July|August|September)
(currentTime ?time&:(> ?time 14))
(currentTime ?time&:(< ?time 17)))
=>
(bind ?class (New Notification))
(call ?class SendNotification ?p))
```

The **defrule** checks in the Jess working memory for facts `PersonLocation` with slot `generalLocation` with value `inBuilding`, and stores the values of the slots `personName` and `buildingName`, in variables `?p` and `?b`, respectively. Then, it checks for facts `Building` with a slot `name` with the same value of the slot `buildingName`, a slot `location` with value inUT, and a slot `temperature` with a value higher than 30 degrees. Finally, it checks for a fact `TemporalInfo` with a slot `months` with value `May` or `June` or `July` or `August` or `September`, and a slot `currentTime` with a value between 14 and 17.

If the engine finds all these facts, it executes the RHS of the rule that creates an object named `?class` by instantiating the `Notification` class and calls a method `SendNotification` on this object in order to notify `?p` (i.e., all the persons that have location in a building of the UT with a temperature higher than 30 degrees).

The clauses **Scope** `(`**Select** `(building.*, build, build.inUT); b))` and **Upon** `EnterTrue (b.temperature > 30)` have been mapped onto the slots `(location inUT)` and `(temperature ?temp&:(> ?temp 30))` of the fact `Building` in the LHS of the **defrule**.

The **When** clause has been mapped onto the slots `(currentTime ?time&:(> ?time 14))` and `(currentTime ?time&:(< ?time 17))` of the fact `TemporalInfo` in the LHS.

The **Do** clause has been mapped onto the RHS of the **defrule**, but the clause **Select** `(person.*, p, p.InBuilding(b))` corresponds to the following code in the LHS:

```
(PersonLocation (personName ?p)(generalLocation inBuilding)
(buildingName ?b1))
(Building (name ?b2&:(eq ?b2 ?b1))
(location inUT)(temperature ?temp&:(> ?temp 30)))
```

Finally, the lifetime **from <**`May`**> to <**`September`**>** has been mapped onto the slot (`month May|June|July|August|September`) of the fact `TemporalInfo` in the LHS.

7.2.4 Example 4

"All persons in the Zilverling building should be notified when there is a presentation in the building that is interesting for them".

We have expressed this scenario by using the following ECA rule (see section 3.3.4):

If <There is a presentation in the Zilverling building AND there are persons in the building interested in this presentation> *then* <Notify (these persons), "This presentation may be interesting for you."">

Figure 32 shows the general mapping from the ECA-DL information model onto Jess `deftemplate`s.

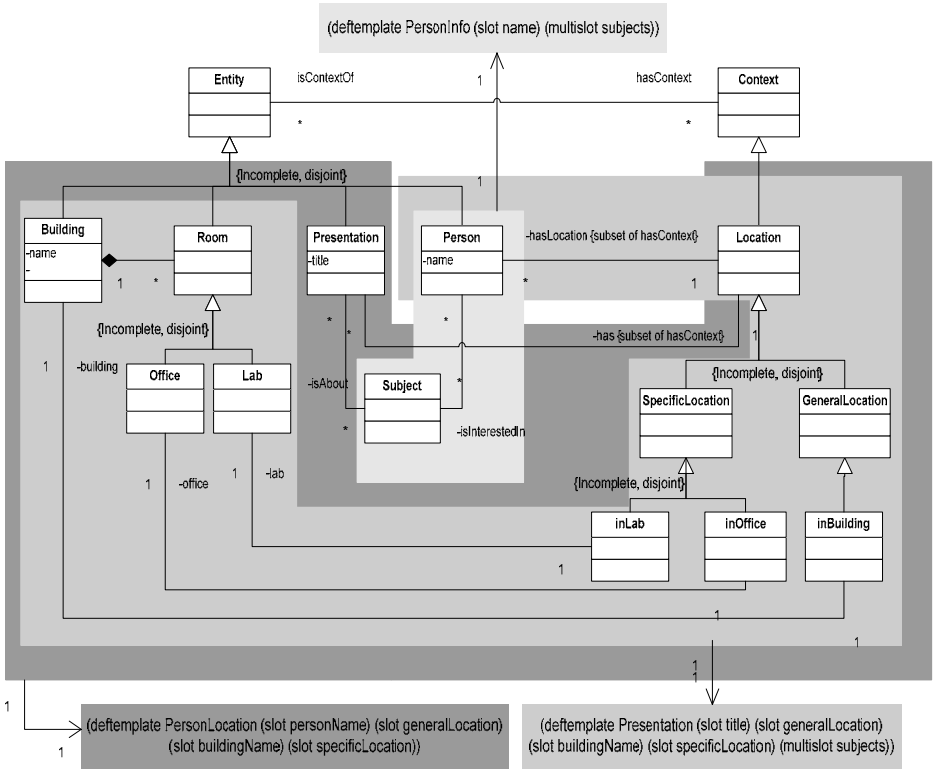

Figure 32 – General mapping of Example 4

Figure 32 depicts the association hasLocation between the entity Person and the context Location. This association has been mapped onto the PersonLocation `deftemplate` used in Examples 1 and 3.

Figure 32 also depicts the associations Presentation has Location and Presentation isAbout Subject, which are both mapped on a `deftemplate` called Presentation. A Presentation has a title (slot title of the `deftemplate`) and a generalLocation (slot generalLocation), which is inBuilding in this example. In this case, Building has a name (slot buildingName). Presentation has also a SpecificLocation (slot specificLocation in the template) and is about one or more subjects (multislot subjects).

The last association depicted in Figure 32 is Person isInterestedIn Subject, which has been mapped onto a `deftemplate` called PersonInfo with a slot name and a multislot subjects.

We have used the following ECA-DL rule in order to express the considered ECA rule:

```
Scope (Select (persons.*, pers, pers.inBuilding.Zilverling);
p)
{
Upon EnterTrue (presentation.inBuilding.Zilverling)
```

```
When p.isInterestedIn(presentation.subject.*)
Do Notify(p), “This presentation maybe is interesting for
you.”
Always
}
```

This ECA-DL rule has been mapped onto the following Jess rule:

```
(defrule example4
(Presentation (title ?t)(generalLocation
inBuilding)(buildingName Zilverling)(subjects ?s1))
(PersonLocation (personName ?p1)
(generalLocation inBuilding)(buildingName Zilverling))
(PersonInfo (name ?p2&:(eq ?p2 ?p1)))
(subjects $?s2&:(member$ ?s1 $?s2)))
=>
(bind ?class (New Notification))
(call ?class SendNotification ?p2 ?t))
```

The **defrule** checks in the Jess working memory for facts `Presentation` with a slot `generalLocation` with value `inBuilding`, a slot `buildingName` with value `Zilverling`, and stores the values of the slots `title` and `subjects` in variables `?t` and `?s1`, respectively. This corresponds to the **Upon** clause of the ECA-DL rule. Then, the **defrule** checks for facts `PersonLocation` with a slot `generalLocation` with value `inBuilding`, a slot `buildingName` with value `Zilverling`, and stores the value of the slot `personName` in variable `?p1`. This is meant to select all persons in the Zilverling building and store their names in variable `?p1`. In this way we have implemented the clauses **Scope** (**Select** `(persons.*, pers, pers.inBuilding.Zilverling); p)` of the ECA-DL rule. Finally, the Jess rule checks for facts `PersonInfo` with a slot `name` with the same value stored in `?p1`, and a slot `subjects` with a value that is contained in `?s1`. This is meant to select all the persons in the Zilverling building that are interested in the same subjects of the presentation. In this way we have implemented the **When** clause of the ECA-DL rule.

If the engine finds all these facts, it executes the RHS of the rule that creates an object named `?class` by instantiating the `Notification` class and calls the method `SendNotification` on this object in order to notify `?p2` (i.e., all persons located in the Zilverling building with interest in the subjects of the presentation) that the presentation with title `?t` may be interesting for them.

7.2.5 Example 5

“When a student is in a meeting with his/her supervisor(s), the meeting time should be counted”.

We have expressed this scenario by using the following ECA rule (see section 3.3.5):

If <Student is in meeting AND this student shares the meeting with his/her supervisor(s)> *then* <Count meeting time>

Figure 33 shows the general mapping from the ECA-DL information model onto Jess `deftemplate`s.

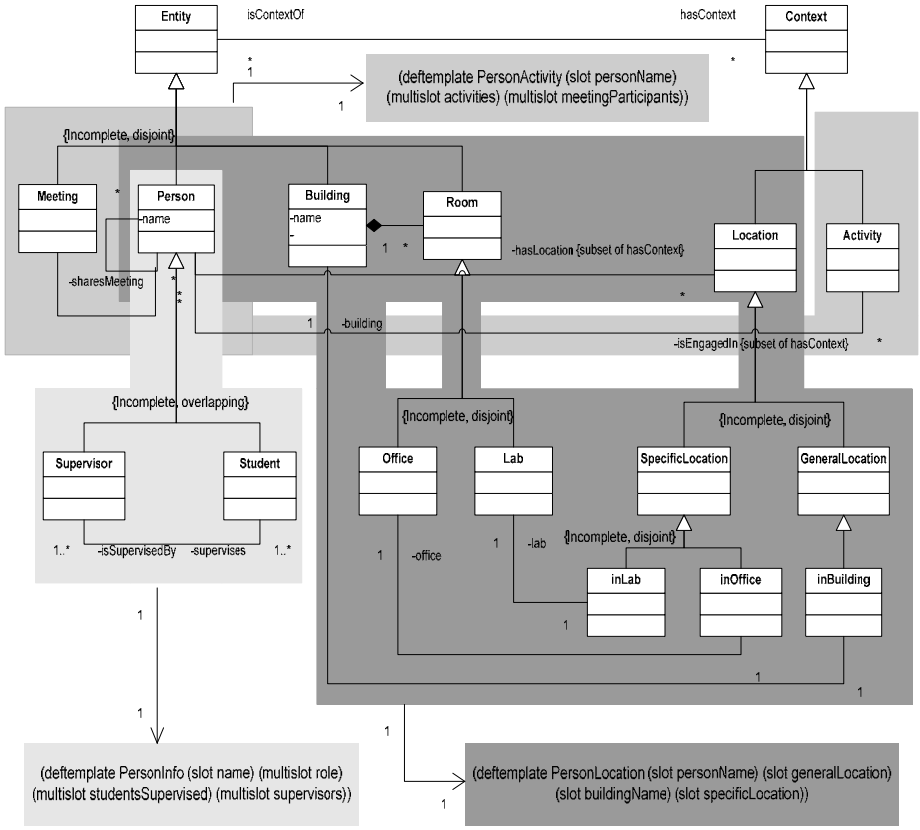

Figure 33 – General mapping of Example 5

Figure 33 depicts again the association hasLocation between the entity Person and the context Location, which has been mapped on the PersonLocation `deftemplate` of Examples 1, 3 and 5. The associations Person isEngagedIn Activity and Person sharesMeeting with another Person have been mapped onto the PersonActivity `deftemplate` of Example 2.

In this example, a Person may be a Supervisor or/and a Student (a supervisor can be a student). We have mapped this information onto the PersonInfo `deftemplate` of Example 4 modified with the addition of new slots. Since a person may be a supervisor or/and a student, we have defined a multislot role, which may have one or both the values student and supervisor. If a Person is Supervisor, we need to know the students supervised (multislot studentsSupervised) and if person is a Student we need to know the supervisors (multislot supervisors).

We have used the two following ECA-DL rules in order to express the considered ECA rule:

```
Scope (Select (student.*, st, st.inBuilding.Zilverling);
stud)
{
Upon EnterTrue (stud.inMeeting)
When stud.sharesMeeting (Select (supervisors.*, super,
superv.stud))
Do StartCountMeetingHours
Always
}
```

```
Scope (Select (student.*, st, st.inBuilding.Zilverling);
stud)
{
Upon TrueToFalse (stud.inMeeting)
When stud.sharesMeeting (Select (supervisors.*, super,
superv.stud))
Do StopCountMeetingHours
Always
}
```

These ECA-DL rules have been mapped on the following Jess rules, where we assume unique identification of persons by their names (no homonymous) in order to improve readability:

```
(defrule firstRuleExample5
(PersonInfo (name ?st1)(role student)(supervisors $?super1))
(PersonLocation (personName ?st2&:(eq ?st2 ?st1))
(generalLocation inBuilding)(buildingName Zilverling))
(PersonInfo (name ?super2&:(member$ ?super2 $?super1))
(role supervisor))
(PersonActivity (personName ?st3&:(eq ?st3 ?st2))
(activities inMeeting)
(meetingParticipants $?part&:(member$ ?super2 $?part)))
=>
(call MeetingHours StartCountMeetingHours ?st3 $?part))

(defrule secondRuleExample5
(PersonInfo (name ?st1)(role student)(supervisors $?super1))
(PersonLocation (personName ?st2&:(eq ?st2 ?st1))
(generalLocation inBuilding)(buildingName Zilverling))
(PersonInfo (name ?super2&:(member$ ?super2 $?super1))
(role supervisor))
(PersonActivity (personName ?st3&:(eq ?st3 ?st2))
(activities ~inMeeting)
(meetingParticipants $?part&:(member$ ?super2 $?part)))
=>
(call MeetingHours StopCountMeetingHours ?st3 $?part))
```

The first **defrule** checks in the Jess working memory for facts `PersonInfo` with a slot `role` with value `student`, and stores the values of the slot `name` and the multislot `supervisors`, respectively, in the `?st1` variable and in the `$?super1` list. Then, it checks for facts `PersonLocation` with a slot `personName` with the same value of the slot `name` of `PersonInfo`, a slot `generalLocation` with value `inBuilding`, and a slot `buildingName` with value `Zilverling`. This is meant to select all the students in the Zilverling building and to store the name of each of these students in a `?st1` variable, and the names of their supervisors in a `$?super1` list. In this way we have mapped the clauses **Scope** `(`**Select** `(student.*, st, st.inBuilding.Zilverling); stud)` to Jess.

The first **defrule** also checks for facts `PersonInfo` with a slot `name` that is in the list `$?super1`, and a slot `role` with value `supervisor`. In this way, we are selecting the supervisors of each student in the Zilverling building. This corresponds to the **Select** `(supervisors.*, super, superv.stud)` clause inside the **When** clause of the ECA-DL rule. Moreover, it checks for facts `PersonActivity` that have a slot `personName` with the same value stored in `?st2`, a multislot `activities` with value `inMeeting`, and a multislot

meetingParticipants that contains the value stored in ?super2. This is meant to select all the students in the Zilverling building that are in meeting with their supervisors and to store the name of each of these students in a ?st3 variable, and the names of their supervisors in a $?part list. In this way we have mapped the **When** clause.

Finally, if the engine finds all these facts, it executes the RHS of the rule that calls the method StartCountMeetingHours on the MeetingHours class. StartCountMeetingHours is a static method that accepts two arguments: a String student and a String...supervisors, which is a string with a variable number of elements. The rule calls the StartCountMeetingHours method on the parameters ?st3 and $?part in order to start counting the meeting hours between the student ?st3 and his/her supervisor(s) $?part. Since static methods do not support multiple instances, we can have only one instance of the StartCountMeetingHours method. This means that only one application at a time can use the MeetingHours class, i.e., if we have a rule that starts counting the time of the meeting between a student and his/her supervisor(s), it has to be followed by another rule that stops counting this time before another meeting time starts to be counted. For this reason, we assume that all meetings between a student and his/her supervisor(s) take place in a dedicated room, therefore we can have only one meeting at a time. In this way we have implemented the **Do** clause.

The second **defrule** proceeds in the same way as the first one with the only difference that it checks for facts PersonActivity with slot activities with value ~inMeeting to execute the method StopCountMeetingHours.

7.3 Mapping Guidelines

From the previous examples shown above, we can see that similar approach has been taken in all examples to map information models and ECA-DL rules onto Jess. In the following sections, we generalize this approach in order to define guidelines for the design of these mappings. These guidelines should be used as input for the automated translation of ECA-DL rules to Jess rules that can be executed by a Controller component built based on Jess.

7.3.1 General guidelines

In the process of developing a context-aware application with the support of a Controller component, we should specify the application's reactive behaviors as ECA rules. These rules refer to an information model composed by entities, context, and relationships between each entity and its context. A context-aware application should consistently define the information models in order to represent the reactive behaviors of applications in a unique and consistent way.

In our examples we have mapped information models onto Jess deftemplates. We consider this as the static part of the mapping, where we map entities, context, and relationships onto static structures (deftemplates). These structures need to be defined before asserting any facts in the working memory of the Jess rule engine.

Objects in ECA-DL (instances of the classes represented in the information models) are mapped on facts in Jess, which are assertions based on the

deftemplates . One or more objects may be mapped onto a single fact. Analogously to instantiating a class in object orientation, we can assert facts based on a deftemplate in the working memory. The operation to instantiate objects in ECA-DL corresponds to the operation to assert facts in Jess. ECA-DL objects are created in an object base, while facts are asserted in the Jess working memory.

ECA-DL rules can be mapped onto Jess rules. ECA-DL rules are based on information models. Analogously, Jess rules refer to deftemplates. ECA-DL rules use objects that are instances of entity and context classes of the corresponding information model. Likewise, defrule constructs in Jess use facts asserted on previously defined deftemplates.

The mapping between information models and deftemplates is characterized by a strong similarity to mappings object models onto relational models, like in relational databases. Information models can be thought as Entity-Relationship (ER) diagrams of a relational database, as well as deftemplates can be thought as tables of this database. Particularly, deftemplate names are like table names of the database, slots names are like columns names, and slots values are like records.

7.3.2 Guidelines for mapping information models

This section provides specific guidelines on how to map classes and relationships of information models to Jess deftemplates. From the discussed examples we have identified the following guidelines:

- Each of the main associations between classes, especially associations that are subset of hasContext, can be mapped onto a deftemplate with a name that properly expresses the purpose of the association. For example, the association Person hasLocation has been mapped on a deftemplate named PersonLocation.

- Sometimes, multiple related associations can be mapped onto a single deftemplate. For example, the association Person isEngagedIn Activity, which is subset of hasContext, and the association Person sharesMeeting(Person) have been mapped onto the PersonActivity deftemplate with slots personName, activities and meetingParticipants. Actually, a possible value of the slot activities is InMeeting, in which case the Meeting can be shared between two or more participants.

- Once we have mapped an association onto a suitable deftemplate, we need to create slots and multislots of this deftemplate. Particularly:

  - If a class holds attributes, each attribute is mapped onto one slot (or multislot) of the deftemplate. For example, class Person has attribute name and we have mapped it onto a slot personName.

  - If a class does not hold attributes, it is directly mapped onto a slot (or multislot). For example, class Temperature has no attributes and we have mapped it onto a slot temperature.

  - Disjoint children classes can be mapped onto a different slot for each of them, or onto alternative values of a single slot. For example, class Location has two disjoint children classes SpecificLocation and GeneralLocation that we have mapped onto two different slots. Class

SpecificLocation has two disjoint children classes InOffice and InLab that we have mapped onto alternative values of the specificLocation slot.

- Overlapping children classes can be mapped onto a single multislot. For example, a person can be Student and/or Supervisor, therefore we have mapped this information onto a multislot role, which can have values supervisor and/or student.

- Associations between children classes can be mapped onto a different slot (or multislot) for each one of them. For example, the associations Supervisor supervises Student and Student isSupervisedBy Supervisor, have been mapped, respectively, onto multislots studentsSupervised and supervisors.

➢ The cardinality of associations considerably influences the mapping. Particularly:

- Cardinality 1..1 is mapped onto one slot that must hold a value different from the `null` value;

- Cardinality 0..1 is mapped onto one slot that holds a value that could be the `null` value;

- Cardinality 1..* is mapped onto one multislot that must hold at least one value, i.e., the multislot cannot be empty (hold the `null` value);

- Cardinality 0..* is mapped onto one multislot that may hold multiple values or be empty (it may hold the `null` value);

- Cardinality *..* is mapped onto a multislot that may hold multiple values, or onto a slot that may hold a range of values defined using some logical predicate, such as "<" or ">".

7.3.3 Guidelines for mapping rules

This section provides guidelines on how to map ECA-DL rules onto Jess `defrule`s. From the discussed examples we have deduced the following guidelines:

➢ One ECA-DL rule is usually mapped onto one Jess `defrule`.

➢ **`Upon`** and **`When`** clauses of an ECA-DL rule (respectively, events and conditions that trigger the rule), are mapped onto the LHS of the corresponding `defrule`.

➢ The **`Do`** clause of an ECA-DL rule (the action to be executed) is mapped onto the RHS of the corresponding `defrule`. If there is a **`Select`** clause inside this **`Do`** clause, the **`Select`** clause has to be mapped onto the LHS of the `defrule`.

➢ **`Scope`** and **`Select`** clauses are always mapped onto the LHS of the `defrule` by using variables assignments and function like `eq`, `neq`, `member$`, for checking for correspondences between these variables.

➢ The lifetime of an ECA-DL rule can be mapped onto the LHS of the `defrule` by using a suitable `deftemplate`. When the lifetime is **`Always`** we do not need a corresponding construct in Jess, since without a specific instruction the engine always fires the applicable rules.

➢ The logical connectives AND, OR of an ECA-DL rule are mapped onto the conditional elements `and`, `or` of the Jess language.

➢ The logical connective NOT of an ECA-DL rule is mapped onto the logic operator ~ of the Jess language. We cannot use the conditional element `not` to define this, since `not` matches the absence of a fact, not the absence of a single value. For example, the fact `(PersonState (personName Laura)(state isWithFriends))` has been mapped onto `Laura.isWithFriends`. If we want to define the condition that Laura is not with friends, in ECA-DL we use `NOT Laura.isWithFriends`, while in Jess we use `(PersonState (personName Laura)(state ~isWithFriends))`, which is different from `(not(PersonState (personName Laura)(state isWithFriends)))`.

7.4 Concluding Remarks

We have produced guidelines for identifying patterns of mappings from ECA-DL onto Jess in examples that are relevant to context-aware applications. For developing large applications there is the need to extend and improve the information models that we have proposed with new types of context information and entities.

In the guidelines that we have proposed we have not yet looked at how to express the *unknown state* (see section 3.1.2) in Jess. Actually, the development of the ECA-DL language is still ongoing work and the unknown state has not been completely specified yet. Therefore, we did not consider this state in our approach.

Moreover, we have not investigated the details on how to map the lifetime construct of ECA-DL rules to Jess. We have presented an example of mapping of the lifetime by using `deftemplate`s in Jess, which is an indication of how this specific mapping can be defined.

Finally, concerning the maintenance of our mapping model, changes in the information models should be reflected in the `deftemplate`s structures in Jess. Therefore, in case of changes, there is the need to update `deftemplate`s and slots in the working memory of the Jess rule engine. Ideally, the maintenance of the working memory should be automated. The automatic translation of the information models to Jess `deftemplate`s (especially at runtime) is a topic for further investigation.

8 Conclusions

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

This chapter is further structured as follows: Section 8.1 presents our general conclusions and summarizes the main contributions of this thesis, and section 8.2 identifies points for future work.

8.1 General Conclusions

We have investigated the mapping of ECA-DL rules onto a well-known tool for developing rule-based system, namely Jess, and we have defined guidelines for performing these mappings. This work is an important part of the design and implementation of a controlling service in the scope of the AWARENESS project. Our efforts included: (i) the study of the AWARENESS project goals, (ii) a literature survey in context-awareness, (iii) an extensive study of rule-based systems, (iv) the definition of criteria in order to choose a tool for developing rule-based systems, (v) the extensive study of the chosen tool (Jess) and (vi) the definition of guidelines to allow ECA-DL rules to be mapped to Jess.

We have explored the benefits of using Event-Control-Action (ECA) pattern, since it provides a high level structure that helps in the design of context-aware applications. This pattern divides the tasks of gathering and processing context information (*Event* module), from tasks of triggering actions in response to context changes (*Action* module) under the control of an application behavior description (*Control* module), in which reactive context-aware application behaviors are described in terms of ECA rules. This separation of concerns effectively enables the distribution of responsibilities in context-aware applications.

The ECA pattern reflects the reactive nature of context-aware applications, whose behaviors can be expressed in ECA rules. In order to facilitate the execution of ECA rules by available technologies, we have made use of a specific language to define ECA rules. This language has been developed in the scope of the AWARENESS project and is coined ECA Domain-specific Language (ECA-DL).

In order to find a suitable technology to execute ECA-DL rules, we have extensively studied rule-based systems. Rule-based systems emulate human expertise in well-defined problem domains by using a knowledge base expressed in terms of rules. We have also analyzed some well-known tools for developing expert systems: CLIPS, Jess, jDREW, and Mandarax, focusing on their main features and application environments, and giving examples of the supported languages. This analysis has been concluded by comparing the mentioned tools on the light of relevant criteria in order to choose the best alternative for our purposes. The most important criteria in the selection process have been the capability to provide support to ECA rules, and the support of tools and IDEs.

We have applied our criteria on the available tools and we have chosen Jess. After studying Jess's architecture and language, we have exploited its most powerful feature, namely that it can be relatively easily integrated with Java. Although we have discussed simple example applications, these applications are a good starting point for implementing large and complex rule-based systems, since they illustrate the basic capabilities of the Jess language. Furthermore, these example applications illustrate how Jess's Java APIs can be used to support context-aware applications and they demonstrate the suitability and expressiveness of the Jess language to write ECA rules.

The ultimate goal of our work has been to define a mapping from ECA-DL onto Jess, since it allows ECA rules written in ECA-DL to be executed in a robust and powerful environment. Particularly, we have studied some examples of ECA-DL rules in order to identify patterns of mappings to be generalized. An important result of this work has been a set of guidelines for the general mapping of ECA-DL rules and information models to Jess rules and `deftemplate`s. These guidelines should be used as input for the implementation of the Controller component.

Although the design of the mapping from ECA-DL to Jess that we have proposed does not consider all the aspects of the ECA-DL language, it is a significant contribution that should be used as a starting point for future extensions.

8.2 Future Work

We have identified the following topics for further investigation:

- Design and implementation of the complete mapping in order to integrate the Jess rule engine, able to execute ECA rules expressed in ECA-DL, within the AWARENESS infrastructure. The integration should be followed by the development of a prototype for the purpose of demonstrating and validating the controlling service.

- Generalization and automation of the mapping in order to enhance productivity and provide an automatic translation of the information models to Jess `deftemplate`s. Changes in information models should automatically reflect to the `deftemplate`s structures in Jess, at system runtime.

- Improvement of some aspects of the mapping that we have not discussed in this thesis. For example, we have not investigated how to express the unknown state in Jess, and we have not investigated the details on how to support the lifetime construct.

- In order to develop large context-aware applications, it is necessary to extend and improve the context information models in order to incorporate context types that are inline with applications' requirements. Some of the extensions may consider the use of Quality of Context (QoC) [35], which provides metadata to quantify the quality of context information in terms of, e.g., accuracy, probability of correctness, and freshness.

- It would be interesting to provide a mapping from ECA-DL to a generic rule engine model, which is not specific to any particular technology. This generic model could be mapped onto different engines, such as the ones that we have studied in this work, with little effort. In this way the mapping effort concentrates on creating a generic model of an application that can be mapped straightforwardly to specific technologies.

References

[1] Riley, G., CLIPS, A Tool for Building Expert Systems. Available at [http://www.ghg.net/clips/WhatIsCLIPS.html#ExpertSystems].

[2] Wikipedia, the Free Encyclopedia. Available at [http://en.wikipedia.org/wiki].

[3] Cawsey, A., Introduction to Expert Systems. Available at [http://www.cee.hw.ac.uk/~alison/ai3notes/section2_5_1.html].

[4] Cawsey, A., Expert Systems Architecture. Available at [http://www.cee.hw.ac.uk/~alison/ai3notes/subsection2_5_2_1.html#].

[5] Rudolph, G., Some Guidelines For Deciding Whether To Use a Rules Engine. Available at [http://herzberg.ca.sandia.gov/jess/guidelines.shtml].

[7] Zaslavsky, A., Mobile Agents: Can They Assist with Context Awareness? School of Computer Science and Software Engineering, Monash University, Australia.

[8] Moran, T. P., Dourish P., Introduction to This Special Issue on Context-Aware Computing. *Special Issue of Human-Computer Interaction, Volume 16, 2001*, IBM Almaden Research Center, University of California, Irvine.

[9] Dey, A. K., et al., Towards a Better Understanding of Context and Context-Awareness. *Technical Report 99-22*, Georgia Institute of Technology, 1999.

[10] Dockhorn Costa, P., Ferreira Pires, L., Van Sinderen, M., Architectural Support for Mobile Context-Aware Applications.

[11] Dockhorn Costa, P., et al., Functional architecture of the AWARENESS infrastructure.

[12] Freeband AWARENESS project. Available at [http://awareness.freeband.nl].

[13] Riley, G., CLIPS, A Tool for Building Expert Systems. Available at [http://www.ghg.net/clips/WhatIsCLIPS.html].

[14] CLIPS homepage. Available at [http://www.ghg.net/clips/CLIPS.html].

[15] CLIPS Reference Manual, Volume 1, Basic Programming Guide, Version 6.23, June 1st 2005.

[16] Jess homepage. Available at [http://herzberg.ca.sandia.gov/jess/].

[17] Ernest Friedman-Hill, Jess in Action, Rule Based Systems in Java.

[18] jDREW homepage. Available at [http://www.jdrew.org/jDREWebsite/jDREW.html].

[19] RuleML Design homepage. Available at [http://www.ruleml.org/indesign.html].

[20] Mandarax homepage. Available at [http://mandarax.sourceforge.net/].

[21] Dietrich, J., The Mandarax 3.0 Manual, Version December 8th 2003, Institute of Sciences & Technology, Te Kura Putaiao o Hangarau-a-Mohiotanga, Massey University, Palmerston North, New Zealand.

[22] Dietrich, J., Wagner, G., Mandarax + ORYX, An Open-Source Rule Platform, Massey University, New Zealand, Eindhoven University of Technology, The Netherlands.

[23] Freeman Hargis, J., Rule-Based Systems and Identification Trees. Available at [http://ai-depot.com/Tutorial/RuleBased.html].

[24] Buschmann, F., et at.: Pattern-Oriented software architecture: A System of Patterns. John Wiley and Sons, New York, U.S.A. (2001).

[25] Dockhorn Costa, P., Ferreira Pires, L.,Van Sinderen, M., Architectural Patterns for context-Aware Services Platform.

[26] Van Bemmel, J., Dockhorn Costa, P., Widya, I., Paradigm: Event-driven Computing.

[27] Dockhorn Costa, P., et al.: AWARENESS Service Infrastructure. AWARENESS Deliverable D2.1(2004). Available at [http://awareness.freeband.nl.]

[28] Chaur G. Wu, Modeling Rule-Based Systems with EMF, November 2004. Available at [http:// www.eclipse.org/articles/].

[29] Forward Chaining, Wikipedia page. Available at [http://en.wikipedia.org/wiki/Forward_chaining].

[30] Backward Chaining, Wikipedia page. Available at [http://en.wikipedia.org/wiki/Backward_chaining].

[31] Protègè homepage. Available at [http://protege.stanford.edu/].

[32] Eclipse homepage. Available at [http://www.eclipse.org/].

[33] Jess, The Rete Algorithm. Available at [http://herzberg.ca.sandia.gov/jess/docs/70/rete.html#].

[34] The Rule Markup Initiative homepage. Available at [http://www.ruleml.org/].

[35] Buchholz T., Küpper A., Schiffers M., “Quality of Contex, What It Is And Why We Need It”, Proc. of the Workshop of the HP OpenView University Association 2003 (HPOVUA 2003), Geneva, 2003.

[36] Wagner G., How to Design a General Rule Markup Language, Eindhoven University of Technology, The Netherlands.