An Active Functionality Service for Open Distributed Heterogeneous Environments

Fachbereich Informatik
Technische Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades
eines Doktors der Ingenieurswissenschaften (Dr.-Ing.)

von

MSc. Mariano Cilia
aus Mar del Plata, Argentinien

Referenten:
Prof. A. P. Buchmann, PhD, TU-Darmstadt
K. Moody, PhD, University of Cambridge

Tag der Einreichung: 12. Juli 2002
Tag der mündlichen Prüfung: 30. August 2002

Darmstädter Dissertation
D17

Foreword

Modern software development assumes a service-based architecture in which well defined portions of the application logic are offered as autonomous services. These services can either be explicitly invoked or they can react to the occurrence of an event. In the former case we require an infrastructure to locate the service and pass its direction and interface description to the requestor. In the latter case we need a notification mechanism that detects the occurrence of an event, reifies it and sends the corresponding notification to a subscriber that can react to the event.

Reactive mechanisms were introduced in the late '80s in the form of Event-Condition-Action rules in active databases. The goal of active databases was to avoid unnecessary and resource intensive polling in monitoring applications where events are detected and the application reacts to the occurrence of these events. In active relational databases, events were modeled as changes of state of the database, i.e. insert, delete and update operations that could trigger a reaction. In object-oriented systems, more general events were defined: temporal events, both absolute and relative, method invocation events, and user-defined events. Events could be both simple or composite, i.e. they could consist of multiple events composed through the operators of an event algebra. A basic form of active functionality was implemented in many systems and is now standard fare in object-relational database systems in the form of triggers. Distributed applications could also benefit from an active (or reactive) functionality. However, the problems in a distributed environment are more complex because of the lack of a global clock. In addition, widely distributed system, e.g. Internet-based distributed systems, suffer from unpredictable delays and loss of messages. In addition, open systems are confronted with heterogeneity.

In spite of the benefits that could be derived from describing the application logic in the form of event-condition-action rules, previous attempts at unbundling active database systems to provide an active service have failed because the basic problems of distributed and heterogeneous systems were not adequately considered. Mariano Cilia addresses these issues and proposes a well-structured distributed active functionality that is completely decoupled from the database system. The active functionality is

itself based on elementary services, such as simple event detection, event composition, condition evaluation, and action execution, that are combined through the use of the same common notification mechanism. Each component was designed to take into consideration the properties of distributed heterogeneous systems, in particular, the need for specifying the context of an event and an ontology-based context description that allows the conversion between contexts and the semantically correct combination of events originating in different contexts. Events are disseminated as notifications by a publish/subscribe system that is based on the notion of concept-based addressing. Concept-based addressing extends content-based addressing/routing by attaching the context to an event publication and an event subscription. This allows the event broker to consider events originating from different environments in a consistent manner. Elementary services interact at a semantic level using the appropriate vocabulary that is based on a common ontology. The ontology considers various levels: the representation level, the basic infrastructure level, and the domain specific level. This work not only considers the aspects of heterogeneity but also ensures that logical event operators are properly defined in the presence of the uncertainties introduced by network delays and failures.

Previous attempts at unbundling centralized active functionality are discussed along three dimensions: the degree of coupling (tight coupling vs. loose coupling), the degree of distribution (centralized vs. distributed), and the degree of heterogeneity (homogeneous vs. heterogeneous). These are not binary issues, a fact that is best observed in the case of distribution: a distributed system in a local area network will behave differently from a distributed system that is connected through the Internet. Each of the above dimensions is discussed and the previously proposed approaches are analyzed. The discussion presented is well founded and mentions the relevant publications in the various areas, whereby the discussion on unbundling of active functionality and distributed event detection and dissemination can be considered to be comprehensive, while the discussion of heterogeneity (by nature of a large body of work originating in the multidatabase area) concentrates on the relevant approaches that address events in heterogeneous environments but does not cover heterogeneity in federated databases.

The foundation of this research is a component-based architecture. It uses ontologies to integrate events coming from different environments, provides a platform for composition of events originating from distributed heterogeneous sources that deals with partial orderings and the lack of a central clock; and an active service that is conceived as a composition of other elementary services. The ontology is introduced at three different levels of abstraction: the basic representation ontology, which is the basis for domain independent physical representation and contains the necessary information for marshalling and unmarshalling of parameters; the infrastructure-specific ontology that contains all the concepts needed for mapping the active functionality, i.e. event

hierarchy, time notions, notifications, and rule primitives; and the domain-specific ontology that contains all the concepts needed for particular domains, such as, auctions or the services provided in an Internet-enabled car. The rule execution is broken down into its component parts: event detection, event composition, condition evaluation and action execution. A framework is presented in which each basic service interacts with the next relevantßervice through notifications. Relevant means in this context that not every elementary service is needed for each rule execution. For example, a rule triggered by a primitive event must not include the event composition elementary service. Event adapters are used for mapping source-specific events to semantic events through the use of concepts from the ontology and the addition of the proper context information. Each service used in the framework is discussed: the notification service, the alarm service, the timestamp service, the complex event selection service, condition evaluation and filter services, the action execution service, and the repository and ontology services.

A key issue in a distributed active functionality is the composition of events. The base of the event composition service is an adequate notion of time. Mr. Cilia does not try to provide a one-size-fits-all approach but rather abstracts the notion of timestamp. A timestamp mechanism "knows"how to represent timestamps and how to correlate them, i.e. how to determine which timestamp is before or after another. This interface can be provided by the abstract timestamp mechanism. Underneath may be different specializations, such as single clock, 2g precedence or accuracy interval. What time model is used depends in the end on the requirements of the application and whether this application can define a chronon that is large enough to make the time inaccuracy irrelevant. Important is, however, that the underlying infrastructure does not provide a false sense of security. Depending on the time model used, different treatment of events must be provided, in particular, partial order must be dealt with, uncertainty and a window mechanism must be provided to separate the history of events into a stable past that does not change and the unstable past and present that may still be subject to change. Mr. Cilia finally provides event compositors that can deal with the stable and unstable portion of the event history and can insert different policies to deal with failure, selection and uncertainty.

A prototype implementation on top of Hewlett-Packard's Core Service Framework validates the proposed concepts and design. CSF provides all the necessary life cycle functions, such as resolution, initialization, start, stop, reconfiguration and destruction of services. On top of this middleware platform, Mr. Cilia implemented the ECA elementary services, specifically a notification service that exploits concept based addressing in the publish/subscribe mechanism. The ontology service is provided based on the MIX/Mibia prototype developed in a previous dissertation by C. Bornhoevd. Adapters that use the ontologies are provided. All the other base services are implemented (composition, filtering and condition evaluation, action execution, etc.). Based

on this platform, Mr. Cilia shows how the distributed active service can be used. The first application that is discussed is a metaauction system that unifies the auctions of multiple sites in a common front-end and tracks for the user multiple auctions at different sites. This implies making the underlying auction mechanisms compatible, providing the proper domain specific ontologies and contexts for all the components of the auction process. The second application that was developed by Mr. Cilia is a set of services in an Internet-enabled car. The system relies on the fact that every car and driver has a Web-presence (a portal) and makes it possible to personalize a car's settings according to the driver's preferences but also to provide services that can be tailored to the driver's needs. For example, a briefing service to read the driver's e-mail on the way to the office or to present the driver with the day's appointments has been implemented.

Prof. Alejandro P. Buchmann, Ph.D.

Darmstadt, September 2002

Acknowledgements

First of all I'm extremely grateful to Prof. Alex Buchmann whose support and fruitful comments during all these years allowed me to produce the present dissertation. Additionally, it is important to acknowledge his constant unconditional help within and outside the academic boundaries.

Thanks to Dr. Ken Moody for his kindly and detailed review of this thesis and the productive discussions during his visits. Thanks also to Prof. Mezini, Prof. Henhapl and Prof. Hoffmann for their constructive comments on this work.

I would like to thank the German Academic Exchange Service (Deutscher Akademischer Austauschdienst, DAAD) for the scholarship during my stay in Germany. As a newcomer to a foreign land, their helpful assistance saved me much time and effort in dealing with bureaucratic affairs. I also acknowledge the help of the people of the Faculty of Sciences and the International Affairs Office at UNICEN University. These three institutions allowed me to concentrate on the research work without worrying about financial matters.

Special thanks to Christof Bornhövd, Christoph Liebig and Felix Gärtner for the fruitful discussions and constructive comments to the manuscripts of this thesis.

I also want to thank all members of the databases and distributed systems group, my colleagues at ITO and those at the Graduiertenkolleg on e-Commerce for creating such a harmonious working atmosphere. I must express my gratitude to Ming-Chuan Wu, who helped us as soon as we arrived in Darmstadt; Christian Haul, for answering my insistent questions and for the enlightening conversations; Andreas Zeidler who helped me to write the German version of the abstract; and finally Marion Braun who has assisted me with many bureaucratic matters.

I would also like to thank the people of the Hewlett-Packard Laboratories in Palo Alto, especially those of the Software Technology Lab and in particular Fabio Casati, Umesh Dayal, Memhet Sayal, Ming-Chien Shan, and Li-Jie Jin. Also Patrick Goddi, who introduced me to the CoolCar scenario, and David Bell who brought me in contact

with the HP Core Service Framework when it was not still a product. Thanks also to Gerhard Lindemann, at Hewlett-Packard Böblingen and initiator of the Hewlett-Packard German Innovation Center, for providing us with a framework to try some of the ideas presented in this work.

To my family who have supported me at a distance and to my old good friends and to those we made during our stay in Germany. Special thanks to my sister Flavia who also gave me stylistic feedback on an early version of this thesis.

My best thanks to Maria, who showed her understanding and patience by supporting me all these years in this important challenge. Last but not least, to my son Francisco, who inspired me while writing this thesis.

Darmstadt, August 2002
Mariano Cilia

Abstract

Companies simply cannot ignore the fundamental problem that business requirements are changing faster than applications can be created and/or modified. Most of these requirements are in the form of or are related to business rules. Business rules are precise statements that describe, constrain and control the structure, operations and strategy of a business. They may be thought of as small pieces of knowledge about a business domain, and offer a way of encapsulating business semantics and making them explicit. Traditionally, business rules have been scattered, hard-coded and replicated by different applications. As a result, it has been difficult to adapt applications to new requirements quickly. In recent years, there has been a trend to extend database technology with powerful rule-processing capabilities leading to the emergence of so-called active databases (aDBMS for short). They rely on Event-Condition-Action rules (ECA-rules) to provide automatic execution of predefined operations in response to the occurrence of certain events. It has been demonstrated that this powerful technology is particularly convenient for enforcing business rules.

Modern large-scale applications, such as e-commerce, Internet or Intranet applications, enterprise application integration (EAI), and emerging pervasive systems, can effectively benefit from an active mechanism but they also impose new requirements. These applications demand support for: information integration, interaction with different systems or services, and collaboration among applications. In this context and from the active database perspective, events and data come from diverse sources, and the execution of actions and evaluation of conditions may be performed on different systems. Conventional active mechanisms have been designed for centralized systems and they are monolithic making it difficult to extend and adapt. Consequently, active mechanisms must be adapted to meet the requirements imposed by this generation of large-scale heterogeneous applications.

The current trend in the application space is moving away from tightly-coupled systems and towards systems of loosely-coupled, dynamically bound components. In such a context, it seems reasonable to move required active functionality outside the active database system by offering an autonomous service that runs decoupled from the

database, and that can be combined in many different ways and used in a variety of environments. For instance, the unbundling approach follows this idea by unbundling the active mechanism into components that will be "rebundled"(or reused) according to application requirements. However, the unbundling approach is inadequate for distributed environments since the aDBMS components to be reused were not designed to take into account inherent characteristics of distributed environments like message delays, the lack of global time, independent failures, and simultaneity of happenings/events. Additionally, the combination of unbundled components and newly developed ones may lead to misinterpretations if the meaning of terms underlying different components is not shared. Moreover, in a distributed and heterogeneous environment active functionality mechanisms are fed with events coming from heterogeneous sources. These events encapsulate data, which can only be properly interpreted when sufficient context information about its intended meaning is known. In general, this information is left implicit and as a consequence, it is lost when data/events are exchanged across institutional or system boundaries. For this reason, to exchange and process events from independent sources in a semantically meaningful way, explicit information about their semantics in the form of additional metadata is required.

The goal of this thesis is to provide more flexible ECA-rule processing functionality than provided by current aDBMS technology in order to support the requirements of open distributed heterogeneous environments. To satisfy these requirements, the active mechanism is decoupled from the database and implemented as a separate, autonomous and flexible active functionality service. This is materialized as a service-based architecture which is founded on three main pillars: an ontology-based infrastructure, event notifications and service-oriented principles. Flexibility is basically achieved due to the service-oriented architecture where elementary services are accordingly composed in order to process the set of defined rules. Semantically meaningful exchange of data and events among heterogeneous participants is accomplished with the help of the underlying ontology-based infrastructure.

In addition, the thesis presents a clear analysis of the difficulties related to composite event detection in distributed environments. On this basis, a separation of concerns is carried out to resolve the problems in isolation while providing a common framework that facilitates the implementation of event operators (that only concentrates on an operators' logic). Other aspects like, event ordering, transmission delays, multiple simultaneous event instances, etc. are treated by means of configurable policies.

In summary, this work provides a platform where business rules of a new generation of applications can be defined across applications at a higher and common level of abstraction. The platform enhances extensibility and maintainability, and supports an effective adaptation to new business requirements.

Zusammenfassung

Heutzutage kommt auf Firmen das grundlegende Problem zu, dass sich Anforderungen an Geschäftsanwendungen so schnell ändern, dass die Entwicklung und/oder Modifikation von Anwendungen einen extrem hohen Aufwand bedeutet. Üblicherweise werden die Anforderungen in Form von Geschäftsregeln (engl.: *Business Rules*) abgelegt. Geschäftsregeln sind präzise Anweisungen, die die Struktur, Transaktionen und Strategie eines Geschäftes beschreiben, einfordern und überwachen. Sie können angesehen werden als kleine Einheiten formalisierten Wissens über eine Geschäfts-domäne und bieten die Möglichkeit, die Geschäfts-Semantik einzukapseln und diese Semantik explizit zu spezifizieren. In der Vergangenheit sind Geschäftsregeln über verschiedene Anwendungen verteilt, fest eingebaut und repliziert worden. Heutzutage ist es deshalb besonders schwer diese Anwendungen an neue Anforderungen anzupassen. Seit einigen Jahren ist der Trend zu beobachten, Datenbankensysteme um umfangreiche Regelverarbeitungs-Mechanismen zu erweitern, was zum entstehen von aktiven Datenbank-Managementsystemen (aDBMS) führte. Sie verwenden so genannte Ereignis-Bedingung-Aktions Regeln (engl.: *Event-Condition-Action rules, ECA-Rules*) um auf bestimmte, vordefinierte Ereignisse reagieren zu können und mit diesen assoziierte Operationen auszuführen. Der Einsatz dieser mächtigen Technologie hat gezeigt, dass sie sich insbesondere für die Durchsetzung von Regeln für Geschäftsprozesse eignet.

Anwendungen von der Grösse aktueller Geschäftsanwendungen, wie e-Commerce, Internet oder Intranet Anwendungen, Anwendung zur Integration von Geschäftsanwendungen (engl.: *Enterprise Application Integration, EAI*) und aufkommende ubiquitäre Systeme können erheblich von einen aktiven Mechanismus profitieren, aber diese Flexibilität hat auch ihren Preis. Sie erfordert nach Unterstützung bei der Integration von Informationen, Interaktion verschiedener Systeme und Dienste und der Zusammenarbeit unterschiedlicher Anwendungen. Das Problem in diesem Zusammenhang ist, dass aus der Perspektive aktiver Datenbanken Ereignisse und Daten aus verschiedenen Quellen stammen und die Durchführung von Aktionen, sowie die Evaluierung von Bedingungen potenziell in unterschiedlichen Systemen stattfindet. Im Gegensatz dazu sind konventionelle aktive Mechanismen für zentralisierte Systeme entworfen worden und daher

monolithisch. Das macht es schwierig, sie an die verteilten heterogenen Anforderungen aktueller Geschäftsanwendungen anzupassen.

Gegenwärtig ist der Trend zu beobachten, dass immer mehr eng gekoppelte Anwendungen entkoppelt werden und aus lose gekoppelten dynamisch gebundenen Komponenten zusammengefügt werden. In diesem Kontext macht es Sinn, die dazu benötigten aktiven Funktionalitäten aus dem Datenbanksystem auszugliedern und als eigenständigen Dienst anzubieten, der unabhängig vom DBMS funktioniert. Der offensichtliche Vorteil ist, dass er flexibel kombinierbar und in vielen unterschiedlichen Umgebungen einsetzbar ist. Der *Unbundling* Ansatz folgt diese Idee und zerlegt die aktive Funktionalität in Einzelkomponenten, die dann zur Laufzeit gemäß der Anforderungen einer Anwendung passend kombiniert werden sollen.

Leider ist der Unbundling Ansatz für verteilte Systeme ungeeignet, da die verwendeten aDBMS Komponenten nicht entworfen wurden, um die inhärenten Charakteristika von Verteilung in Betracht zu ziehen. Beispiele dafür sind: Verzögerung und Verlust von Nachrichten, fehlende globale Zeit, spontan auftretende Ausfälle oder die unmöglichkeit eine globale Ordnung der Ereignisse zu bestimmen. Ein zusätzliches Problem kann eine unterschiedliche Interpretationsbasis verschiedener aus den aDBMS herausgelöster Komponenten im Zusammenspiel mit neu entwickelten Komponenten sein, die zu Fehlinterpretation von Daten führen kann, wenn die zugrunde liegende Bedeutung von Bezeichnungen unterschiedlich ist. Außerdem werden in einer verteilten und heterogenen Umgebung die Mechanismen für aktive Funktionalität mit Ereignissen bedient, welche ebenfalls aus heterogenen Quellen stammen. Diese Ereignissen kapseln Daten ein, die nur dann geeignet interpretiert werden können, wenn ausreichende Kontextinformationen über die beabsichtigte Bedeutung bekannt sind. Im Allgemeinen ist diese Information implizit und geht verloren, wenn Daten/Ereignisse über Institutions- oder Systemgrenzen hinweg ausgetauscht werden. Aus diesen Grund und um Ereignisse aus unabhängigen Quellen in einer semantisch sinnvollen Weise bearbeiten zu können, müssen zusätzliche Informationen über die explizite Semantik in Form von Metadaten hinzugefügt werden.

Ziel dieser Dissertation ist es flexiblere Mechanismen zur Verarbeitung von ECA-Regeln anzugeben, als sie in konventionellen aDBMS heute zu finden sind, um die Anforderungen von offenen, verteilten, heterogenen Systemen zu unterstützen. Um diesen Anforderungen gerecht zu werden, wird der aktive Mechanismus von der Datenbank entkoppelt und als separater und eigenständiger Dienst implementiert. Gemäß der Strategie, Dienste voneinander zu entkoppeln, wird die Gesamtfunktionalität des Dienstes erst durch die Komposition verschiedener anderer elementarer Dienste ermöglicht. Daher ist es sinnvoll, eine Dienstarchitektur anzugeben, die auf drei zentralen Säulen ruht: Einer Ontologie-basierten Infrastruktur, einem Notifikations-Dienst und einem Dienstorientierten Paradigma. Flexibilität wird im Wesentlichen dadurch erreicht, dass in

dieser Dienst-orientierten Infrastruktur elementare Dienste passend komponiert werden, um die Menge der aktuell definierten Regeln abzuarbeiten. Semantikerhaltender Austausch von Ereignissen/Daten zwischen heterogenen Entitäten wird durch die zugrunde liegende Ontologie-basierte Infrastruktur erreicht.

Zusätzlich analisiert die vorliegende Dissertation detailliert die Probleme komplexer Ereigniserkennung in verteilten Systemen. Auf dieser Basis wurde eine Trennung der Belange (engl.: *separation of concerns*) durchgeführt, um die Probleme jeweils getrennt zu lösen und gleichzeitig auch ein gemeinsames Rahmenwerk aufzubauen das die Implementierung der Ereignis-Operatoren erleichtert. Andere zentrale Aspekte, wie partielle Ordnung der Ereignisse, Verzögerungen von Übertragungen, das simultane Auftreten von Ereignissen, usw., werden durch die Angabe konfigurierbarer Strategien behandelt.

In dieser Arbeit wird eine Plattform definiert auf deren Basis Regeln moderner Geschäftsanwendungen über Anwendungsgrenzen hinweg und auf einer höheren und allgemeineren Ebene der Abstraktion definiert werden können. Die Plattform verbessert die Erweiterbarkeit sowie die Wartbarkeit von Geschäftsanwendungen und unterstützt eine effektive Anpassung an neue Geschäfsanforderungen.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Companies simply cannot ignore the fundamental problem that business requirements are changing faster than applications can be created and/or modified. Most of these requirements are in the form of or are related to business rules. Business rules are precise statements that describe, constrain and control the structure, operations and strategy of a business. They may be thought of as small pieces of knowledge about a business domain. They offer a way of encapsulating business semantics and making them explicit in the same way that databases enable the separation of data from application programs.

Traditionally, business rules have been scattered, hard-coded and replicated by different applications. The lack of a formal approach to the management of business rules and a standard business rule language has made it virtually impossible to create, modify and manage business rules in a flexible way. As a result, it has been difficult to adapt applications to new requirements quickly.

Isolating the business rules from the application code enables developers to easily find and modify the pertinent rule(s) when a policy change is required. This makes it possible to quickly change rules without modifying the rest of the application code, thereby enhancing maintainability.

A first attempt from the database systems was the support of assertions that were in charge of checking constraints [Hammer and Sarin 1978]. In recent years, one of the trends in database technology has focused on extending conventional database systems (DBMS) to enhance their functionality and to accommodate more advanced

applications. One of these enhancements was extending database systems with powerful rule-processing capabilities. These capabilities can be divided into two classes: *deductive*, in which logic programming style rules are used to provide a more powerful user interface than that provided by most database query languages [Ceri et al. 1990]; and *active*, where production style rules are used to provide automatic execution of predefined operations in response to the occurrence of certain events [Dayal et al. 1988; Act-Net Consortium 1996]. The latter is particularly appropriate for enforcing business rules as it has been demonstrated in [Ceri and Widom 1996; Paton 1999]. Database systems enhanced with active capabilities are known as *active databases systems* or aDBMS for short.

In their most general form, active database rules (also known as ECA-rules) consist of three parts:

- **E***vent*: causes the rule to be triggered,
- **C***ondition*: is checked when the rule is triggered, and
- **A***ction*: is executed when the rule is triggered and its condition evaluates true

By means of active database systems, general integrity constraints encoded in applications have been moved in the form of rules into the database system. These rules go beyond key or referential integrity constraints. Active databases support the specification and monitoring of general constraints (rules), they allow flexibility in the time of constraint checking and they provide execution of compensating actions to rectify a constraint violation without rolling back the involved transaction. Additionally, support for external events and actions were introduced mostly to satisfy the requirements of monitoring applications.

As a consequence, applications sharing the same database system and data model can also share business rules. In this way, the business knowledge that was dispersed in many applications in the form of programming code is now represented in the form of rules and managed in a centralized way. Consequently, when business rules change only those affected rules must be modified in the aDBMS.

However, when monitoring external applications, signals must go through the aDBMS to trigger rules even though they do not have a direct relation with the database. Moreover, each active database implementation uses its own (low-level) rule definition dialect making, in some cases, rule definition cumbersome. Additionally, traditional active mechanisms have been designed for centralized systems and are monolithic and tightly integrated, thus making it difficult to extend or adapt them to a new generation of applications.

Modern large-scale applications, such as e-commerce, enterprise application integration (EAI), Internet or Intranet applications, impose new requirements. In these applications, integration of different subsystems and collaboration with partners' applications is of particular interest, since business rules are out of the scope of a single application. Events and data are coming from diverse sources, and the execution of actions and evaluation of conditions may be performed on different systems. Furthermore, events, conditions and actions may not be necessarily directly related to database operations. A similar situation can be seen if considering the trend of pervasive and ubiquitous computing. Many applications in this area are related to context-awareness where actions need to be taken as a response to context changes. This field is characterized as highly decentralized and distributed over a multitude of different (sensing) devices that can be dynamically networked and interact in an event-driven manner. At a glance, this arena can benefit from the active database technology but again, in this kind of applications, no direct relation with a database seems to be needed. This leads to the question of why a full-fledged database system is required when only active functionality and some services of a DBMS are used.

The current trend in the application space is moving away from tightly-coupled systems and towards systems of loosely-coupled, dynamically bound components. In such a context, it seems reasonable to move required active functionality outside the active database system by offering a flexible service that runs decoupled from the database, and that can be combined in many different ways and used in a variety of environments. For this, a component-based architecture seems to be appropriate [Gatziu et al. 1998; Collet et al. 1998], in which an active functionality (ECA-rule) service can be seen as a combination of other components, like complex event detection, condition evaluation, and action execution. Thus, components can be combined and configured according to the required functionality, as proposed by the unbundling approach in the context of aDBMSs [Gatziu et al. 1998; Koschel et al. 1999].

Whether the unbundling approach is realistic for active database systems or not, it is inadequate for distributed environments since aDBMS components to be "rebundled" are designed with a homogeneous, centralized environment in mind. That means that they were not conceived to take into account inherent characteristics of distributed environments like independent failures, message delays, and the lack of a global time. These issues have an impact not only on the composite event detector but also on the semantics of event operators [Liebig et al. 1999], consequently, the reuse (or rebundle) of traditional complex event detection mechanisms in this kind of environment is not viable. Additionally, the combination of unbundled components and newly developed ones may lead to misinterpretations if the meaning of terms underlying different components is not shared.

Moreover, in a distributed and heterogeneous environment active functionality mech-

anisms are fed with events coming from heterogeneous/diverse sources. These events encapsulate data, which can only be properly interpreted when sufficient context information about its intended meaning is known. In general, this information is left implicit and as a consequence, it is lost when data/events are exchanged across institutional or system boundaries. Combining or interpreting data from different sources leads inevitably to problems if the meaning of terms is not shared [Bornhövd 2000]. For this reason, to exchange and process events from independent sources in a semantically meaningful way, explicit information about its semantics in the form of additional metadata is required.

As a result, the difficulties introduced by heterogeneity, and the inherent characteristics imposed by large-scale distributed environments, have an impact on the following issues of active functionality:

- event exchange mechanism,
- event semantics (correct interpretation and use of events), and
- complex event detection.

1.2 Proposed Approach

This work takes active database technology and current active systems as a departure point and analyzes the problems related to their adaptability to distributed and heterogeneous environments. In particular, this section presents the conception of this thesis highlighting the main lineaments that has been followed to achieve the target environment. Figure 1.1 tries to sketch these ideas and the path taken. It begins at the top with an aDBMS representing the departing point, and it ends up with a flexible active functionality service at the bottom. Between those ends, the lineaments followed in this work are illustrated and a brief description is presented below.

As mentioned before, the goal of this work is to provide an active functionality service completely decoupled from the database system. For this reason, and at a conceptual level, the active mechanism was first separated from the database and later unbundled into components. Some of these components were redesigned to satisfy the requirements of open, loosely-coupled, distributed environments (see Figure 1.1 A).

Normally, aDBMSs offer a generic language to specify ECA-rules. This specification language determines what can be defined as a rule and which features or characteristics are available from the active functionality perspective. A generic rule specification is

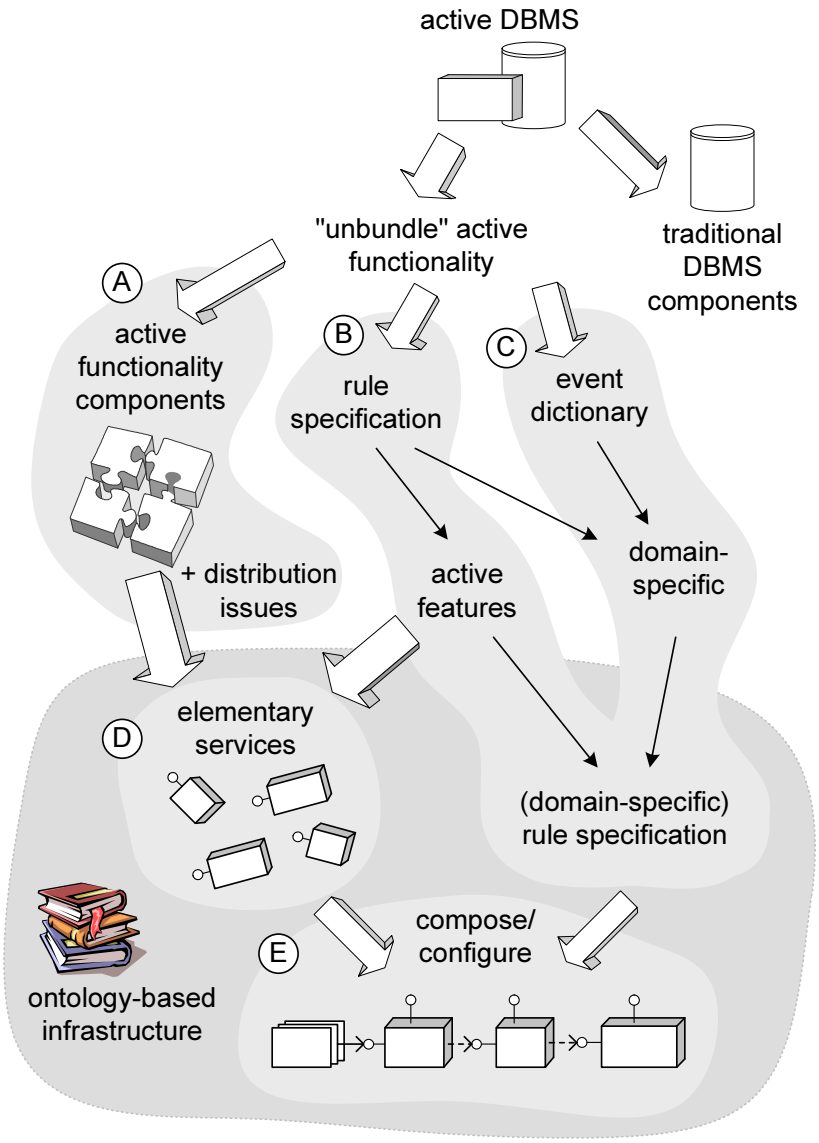

Figure 1.1: Abstract view of the proposed approach

essentially motivated by the use of a generic active functionality mechanism. Trying to satisfy the requirements of dissimilar domains ends up in many cases with the following problem: the expressive power of such a specification language is inversely proportional to its usability. That means that rule specifications are cumbersome to define and in consequence, difficult to understand. It seems convenient to have a rule specification language for each particular domain relying on an intermediate (generic) representation. This rule representation is, in turn, the input for the underlying active

functionality service (see Figure 1.1 B).

In current active systems, events are defined using event types that basically define syntax and structure. However, they do not pay attention to semantics or contextual information. Notice that in an heterogeneous environment, events come from diverse sources where each participant probably adopts different assumptions about data. In such a context, it is essential to use a common vocabulary (or ontology) to be shared among all participants (applications that signal and consume events) allowing them to add contextual information to make these assumptions explicit in order to correctly interpret data and events (see Figure 1.1 C).

With this in mind and now from another perspective, the use of ontologies can be applied not only to represent events but also to describe the terminology that belongs to active functionality. This allows loosely-coupled components in this arena to interact with each other using a common "active-functionality-related" vocabulary (see Figure 1.1 D).

Once a rule is specified and submitted to the active functionality service, selected elementary services/components are dynamically configured according to the rule specification in order to cooperate in its execution (see Figure 1.1 E).

1.3 Contributions of this Thesis

In this work the active functionality field was analyzed with the purpose to offer this useful functionality in other environments by adapting it to support modern applications. On the basis of this analysis, a novel approach of combining ontologies, notifications and services was proposed. Of particular interest was the integral use of ontologies not only for supporting the correct interpretation of heterogeneous data but also for the active functionality service itself. In this way, an ontology-based infrastructure was proposed where notifications (their representation and the way they are addressed), service interfaces, and rule definitions were specified in an infrastructure-specific ontology. This kind of organization clearly separates the terminology related to the problem that is being solved from those related to the active infrastructure.

Because of its conceptual foundation, this architecture promotes extensibility and integration for modern Internet-based applications. Flexibility was basically achieved due to the service-oriented architecture where elementary services are composed in order to process the defined set of rules. These services can be developed by different, independent providers but all conforming to a simple interface, and with a simple and clear task to be carried out. The interaction among elementary services relies on a concept-based (publish/subscribe) notification service. Additionally, this architecture encourages the

easy adaptation of the active service to satisfy new requirements. Particularly, this work provides an extensible platform where the underlying assumptions and the resulting semantics are clearly stated and explicitly defined making its understanding easier. It seems to be an ideal platform, in contrast to one-of-a-kind prototypes, to explore other aspects/topics related with active functionality.

The main benefits of this proposal include the following aspects:

- events from different sources are represented using terms of a common vocabulary (concepts of an ontology) and additional contextual information,
- events are disseminated as notifications by means of a publish/subscribe notification service, that is adequate for distributed environments,
- a concept-based addressing was proposed to empower the way notifications are addressed by maintaining a common and higher-level of abstraction to describe the interests of publishers and subscribers,
- elementary services interact at a semantic level using an appropriate vocabulary,
- rule definition languages can be tailored for different domains relying on a conceptual representation,
- the conceptual rule representation enables the use of a common and generic active service,
- a complex event detection mechanism based on the principles of container and component that is suitable for distributed environments, and
- a clear and isolated definition of the logic of event operators.

This thesis presents a clear analysis of the difficulties involved in complex event detection in distributed environments. Based on these results, a separation of concerns has been carried out to resolve the problems in isolation while providing a common framework in order to make easier the implementation of event operators.

As a final remark, this work provides a flexible active functionality platform that enables adaptability and extensibility in a variety of environments.

1.4 Issues not Addressed in this Thesis

The work in this dissertation concentrates on active functionality, without covering issues related to deductive databases. Techniques for analyzing rules to ensure termination, confluence, and determinism are out of the scope of this work but considered as an important area of future research.

1.5 Organization

The dissertation is organized as follows. **Chapter 2** provides a discussion of the most important aspects and known approaches for adapting/extending centralized active functionality for distributed heterogeneous environments. Basically three aspects are discussed: heterogeneity, distribution and loosely coupled systems.

In **Chapter 3** the conceptual foundation for a flexible and extensible active service for this kind of environment is presented including an overall picture. After that, in **Chapter 4**, the service-based architecture is introduced, describing elementary services that take part in rule processing. Additionally, the role of event adapters is explained in context.

Chapter 5 presents the difficulties for detecting complex events in a distributed environment and describes the approach followed in this work towards a flexible complex event detection mechanism.

Details about the implementation of an active functionality service that follows the principles introduced in previous chapters are presented in **Chapter 6**. This includes the description of the implementation of ontologies, the e-service platform used, as well as the services required for an active service. Besides that, the implementation of some event adapters and other utilities are presented. To present the proposed approach in context, two different case studies are described in **Chapter 7**: online auctions and vehicle personalization.

Finally, in **Chapter 8** the conclusions of this thesis are presented, addressing open issues and future work.

To streamline the presentation, some of the background topics are included as an Appendix that can be consulted at the reader's discretion. **Appendix A** includes the essentials of ECA-rule processing, publish/subscribe messaging and e-services.

Additionally, two other appendices are included in this thesis. **Appendix B** consists of the definition of the ontology that provides the foundation of the infrastructure.

Appendix C contains the ontology definitions associated with the case studies presented along this work. Examples presented throughout this dissertation are related to the online auction scenario.

In the rest of this dissertation, the term ECA-rule is used interchangeable with rule to mean the same thing, as well as, concept with ontology concept, composite events with complex events, and semantic object with MIX-object.

Chapter 2

Related Work

As mentioned before, this work tries to extend traditional active functionality to support loosely-coupled, distributed, heterogeneous environments, as schematized in Figure 2.1. Implications of this expansion and a review of other approaches that address these issues to some degree are discussed in this chapter.

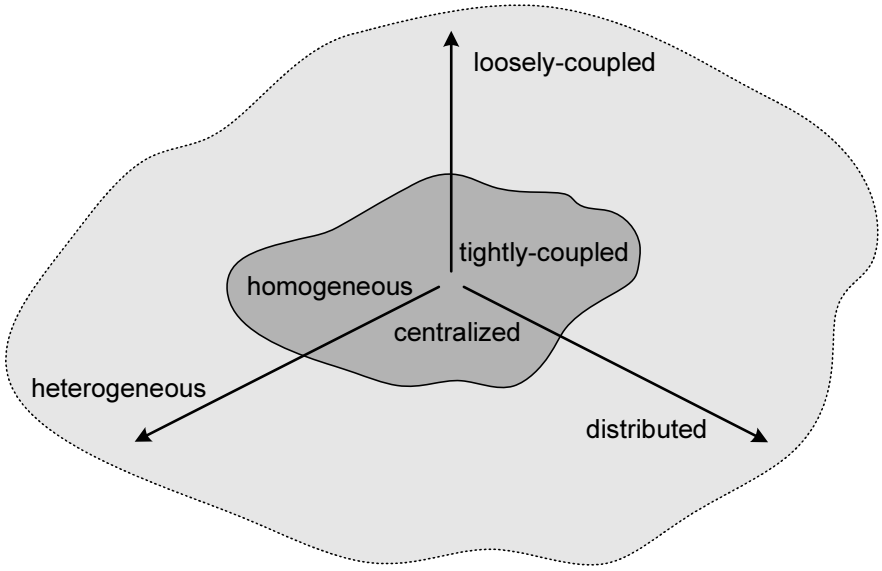

Figure 2.1: Expanding active functionality to support new environments

2.1 Heterogeneity

An information integration system that provides access to data from a multitude of distributed, diverse and autonomous information sources, needs to resolve the heterogeneities between them. Interoperability-based approaches focus on the exchange of meaningful, context-driven data between autonomous systems. From the heterogeneity dimension perspective, there are different levels of heterogeneity (see Figure 2.2): system, syntax, structure and semantics [Ouksel and Sheth 1999]. *System* heterogeneity involves aspects related to the platform (basically operating systems and hardware) and the information systems involved, such as DBMS, data models, system capabilities, etc. At the *syntactic* level, different machine-readable aspects of data representations (also referred as formatting) are considered. The *structure* level deals with representational heterogeneities which involve data modeling and also schematic heterogeneity that appears in structured databases. *Semantic* interoperability tries to support high-level, context-sensitive information requests over heterogeneous information resources, hiding system, syntax and structural heterogeneity. Ontologies are being used to describe information and also as a tool to resolve semantic heterogeneity conflicts [Gruber 1995; Guarino 1997; Goh et al. 1999; Bornhövd and Buchmann 1999; Hakimpour and Geppert 2001].

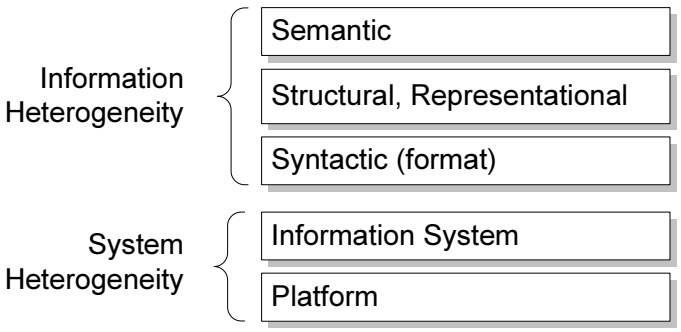

Figure 2.2: Heterogeneity in information systems

From the database perspective, the integration of events coming from heterogeneous sources is comparable to the problems faced in federated multi-database systems. Comprehensive work in this area can be found in [Kim and Seo 1991; Kim et al. 1993; Garcia-Solaco et al. 1996; Hakimpour and Geppert 2001]. These systems consist of a set of autonomous heterogeneous databases where data from each database are accessed using a unified interface. The autonomy of the participating databases is important because responsible institutions want to keep control over local interfaces and still run their local applications. There are basically two approaches for multi-database integration, the tightly and the loosely coupled approach. The first one is based on schema

integration, i.e. it maps the schemas of the participating databases into a conceptual global schema. In this case, applications access data through the global schema. The loosely coupled approach, on the other hand, makes heterogeneity of the participant databases visible to the applications, and provides support for resolving heterogeneity by the applications defining a specific multi-database manipulation language.

Similar to heterogeneity in data, heterogeneity in events (which can be understood as containing event-descriptive data) coming from different sources is also present. In C^{2}offein [Koschel and Lockemann 1998], event sources are encapsulated by means of wrappers. These wrappers map application-specific events into a shared event description defining syntax and structure. [Koschel et al. 1999] propose abstract connectors to hide a set of heterogeneous components, making invisible the fact that different event sources exist.

In [Chakravarthy et al. 1999] the problem of heterogeneity in ECA rule processing systems is treated at the system level, in particular, they use CORBA to integrate different systems, implemented with different programing languages. Here events are defined using CORBA IDLs and no other high-level aspects of data integration are discussed.

Other projects, like NAOS [Collet et al. 1998], try to extend active capabilities towards integration of cooperative and heterogeneous applications but they do not explain how this is achieved. In the same way, in [Bates et al. 1998] it is proposed to use events as a means of communication between heterogeneous components that were not designed to interoperate. In this article, the problem of event interpretation is mentioned but not treated in depth.

To integrate data/events from different sources, the approaches mentioned above concentrate on structural aspects and assume global knowledge by the database administrator (DBA) or the application developer of the assumed semantics of all relevant data and events. This assumption is unrealistic in a large and very dynamic environment like the Internet. Notice that data/events must be compared or correlated externally from the source generating the event. Events containing date or price attributes require explicit knowledge about modeling assumptions (regarding format or currency) to correctly interpret them. Moreover, consumers and producers of events may be previously unknown or they may not share the same political or cultural context, therefore a common structural and semantic representation basis (common vocabulary) of the events involved is necessary for their correct interpretation and use [Bornhövd and Buchmann 1999]. Consider an online auction scenario where participants may have/assume different contexts. Here a common vocabulary is mandatory (normally established using categories of items) and because of its global scope, representations of all the descriptive information require context information, such as date and time format, metric

system, currency, etc. to correctly interpret data. Such an approach is presented in [Bornhövd et al. 2000].

2.2 Unbundling Active Database Functionality into Reusable Components

Active database functionality developed for a particular DBMS is becoming part of a large monolithic piece of software (the DBMS itself). Monolithic software is difficult to extend and adapt. Moreover, active functionality tightly coupled to a concrete database system hinders its adaptation to today's Internet applications, such as e-commerce, where heterogeneity and distribution play a significant role but are not directly supported by traditional (active) database systems [Koschel et al. 1999].

Another weakness of tightly coupled aDBMSs is that active functionality cannot be used on its own without the full data management functionality. However, active functionality is also needed in applications that require no database functionality at all, or that require only simple persistence support. As a consequence, active functionality should be offered not only as part of the DBMS, but also as a separate service that can be combined with other services to support, among others, Internet-scale applications.

Unbundling is the activity of decomposing systems into a set of reusable components and their relationships [Geppert and Dittrich 1998]. Unbundling active databases consists of separating the active part from active DBMSs and breaking it up into components providing services like event detection, rule definition, rule management, and execution of ECA rules on the one hand and persistence, transaction management and query processing services on the other [Gatziu et al. 1998]. Afterwards, only necessary components can be rebundled in order to provide the required functionality. A separation of active and conventional database functionality would allow the use of active capabilities depending on given application needs without the overhead of components that are not needed.

Other projects follow a similar approach. For instance, the C^{2}offein project [Koschel et al. 1997; Koschel and Lockemann 1998] proposes a widely configurable service set of active functionality in CORBA-based heterogeneous, distributed systems. The system is configurable with respect to service features and interaction protocols, distribution parameters, etc.

FRAMBOISE [Fritschi et al. 1997; Fritschi et al. 1998] introduced a construction system for the development of ECA-services that are decoupled from a particular DBMS.

It includes tools to specify and generate adapters that can be applied in conjunction with traditional DBMSs.

In [Collet et al. 1998] the authors defined an event model with several dimensions that characterize event definition, detection, production and notification trying to extend the event concepts to handle different event management semantics, to consider the distribution dimension, and to open it to the integration of cooperative and heterogeneous applications. Collet [2000] sees a database system as an open platform comprised of cooperating, adaptable and extensible services. Here the NODS project is presented and a description of a persistence, an event and rule service are included. Vargas-Solar [2000] follows this approach by specifying an event service that generates event managers as software components customizable according to application requirements and to environment. These approaches do not present with sufficient detail how components cooperate and how problems related to event composition in distributed environments are resolved.

It should be noted that breaking the active service into medium-grained components makes it possible to combine them in a variety of ways. For instance, some components can be omitted or new components can be incorporated for a particular scenario. As a collateral effect, the communication among these components could have a negative impact on performance. Thus, there is a trade-off between performance and flexibility.

From our point of view, unbundling active functionality from a concrete system and then rebundling the corresponding components in an open distributed environment is not feasible. Unbundling in this context means to give up the "closed world" assumption that traditionally underlies a DBMS. Inherent characteristics of open distributed environments impose new requirements that were not considered in centralized environments, such as the lack of a global time, independent failures of nodes or communication channels, message delays, etc.

By means of a central clock, active databases timestamp events with the purpose of ordering them to try to detect complex situations. But in contrast with a centralized system where events can be totally ordered, in distributed environments this is not possible due to the nature/existence of concurrent/simultaneous happenings and the potential message delays.

The consideration of these characteristics has a direct impact on the event detector [Liebig et al. 1999], which is the essential component of an aDBMS [Buchmann 1999]. Consequently, it would not be feasible to reuse components taken from centralized aDBMSs since they ignore relevant aspects of the new environment. In addition, the semantics of operators and consumption modes are hard-wired in the code of existing aDBMS event detectors.

In the unbundling approaches mentioned above a generic architecture is defined, where the interaction among components is exposed. However, there may be difficulties when integrating unbundled and newly developed (autonomous) components. Notice that the meaning of terms employed by different components can lead to misinterpretations if a common semantic basis, i.e., vocabulary, is not shared. For instance, just consider the case of timestamps that are exchanged among components. Probably, these components have different implicit assumptions (format, time zone, etc.) that are not declared but required to correctly interpret their timestamps.

Buchmann and Liebig [1999] discuss crosseffects and potential incompatibilities in order to provide a foundation for a configurable middleware platform that combines selected features of active, real-time and distributed object systems.

2.3 Distribution

Moving centralized active functionality to open distributed environments leads to two main issues. The first one is how event occurrences are efficiently delivered/disseminated to the proper consumers. The second one relates to the question of how complex event detection is performed in an environment characterized by the lack of a global time (necessary to give an order occurrence of events), independent failures of nodes and communication channels and message delays. These two main issues are discussed below.

2.3.1 Event Dissemination

In a distributed environment events must be propagated to all interested consumers. For this purpose, event notification services, or notification services for short, are widely used. There are several research projects (e.g. SIENA [Carzaniga 1998], REBECA [Mühl 2001], CEA [Bacon et al. 1998], JEDI [Cugola et al. 1998], READY [Gruber et al. 1999], ELVIN [Segall and Arnold 1997]), standard specifications [Object Management Group 1997; Hapner et al. 1999] and products [TIBCO; IBM; Fiorano; Talarian; SonicSoftware; SpiritSoft] that focus on different aspects of data dissemination.

In the distributed object platform CORBA, the event service [Object Management Group 1997] was introduced to provide a mechanism for decoupled, asynchronous interaction between CORBA objects. In this context, the event channel acts as a mediator between suppliers and consumers of events. To overcome deficiencies of this service specification, the notification service [Object Management Group 1998] was proposed

as a major extension which additionally provides support for quality of service specifications and which also introduces event filtering.

The Java Message Service (JMS) [Hapner et al. 1999] provides the Java technology platform with the the ability to process asynchronous messages. JMS was originally developed to provide a standard/common java interface (API) to legacy Message Oriented Middleware (MOM) products like IBM MQ-Series [IBM] or TIB/Rendezvous [TIBCO]. Nowadays many companies are offering a new generation of lightweight pure Java messaging service implementations like FioranoMQ [Fiorano], SmartSockets for JMS [Talarian], SonicMQ [SonicSoftware], Spirit Lite [SpiritSoft]. In the JMS model, clients of a message service send and receive messages through a provider that is responsible for delivering messages. In this way, the JMS API provides portability of java code, so the underlying messaging service can be replaced without affecting programming code. The JMS provides two models for messaging among clients: point-to-point (using a queue) and publish/subscribe (by means of topics).

JMS has been a part of the Java Enterprise Edition (J2EE) [Sun Microsystems 2001] suite of java technologies since its origin but it was incorporated as an integral part of the Enterprise Java Beans (EJB) component model in the EJB 2.0 specification [DeMichiel et al. 2001]. Up to this point there were no formal means by which EJB components could make use of this technology. This specification extends the EJB component model to incorporate a new bean type, known as message-driven bean, which acts as a message consumer providing asynchrony to EJB-based applications. A message bean is associated to a JMS topic or queue and receives corresponding messages sent by other beans or java programs, or by other non-java message producers. The integration of JMS and EJB allows enterprise beans to participate in loosely connected systems.

In recent years academia and industry have concentrated on publish/subscribe mechanisms[1] because they offer loosely coupled exchange of asynchronous notifications, facilitating extensibility and flexibility.

The channel model has evolved to a more flexible subscription mechanism, known as subject-based, where a subject is attached to each notification [Oki et al. 1993][TIBCO]. Subject names consist of one or more elements (usually a string) organized in a tree by means of a dot notation. Subject-based addressing features a set of rules that defines a uniform name space for messages and their destinations. This approach is inflexible if changes to the subject organization are required, implying fixes in all participant applications.

To improve expressiveness of the subscription model the content-based approach was proposed where the whole content of a notification can be used for subscriptions. This

[1] A description of publish/subscribe mechanisms can be found in Appendix A – Section A.2.

approach is more flexible but it requires a more complex infrastructure [Carzaniga et al. 1999]. Many projects under this category concentrate on scalability issues on wide-area networks and on efficient algorithms and techniques for matching and routing notifications reducing network traffic [Carzaniga et al. 2000; Carzaniga et al. 2001; Aguilera et al. 1999; Banavar et al. 1999; Opyrchal et al. 2000; Mühl 2001; Mühl et al. 2002; Fabret et al. 2001]. Most of these approaches use simple boolean expressions as subscription patterns since more powerful expressions cannot be treated.

2.3.2 Detecting Global Composite Events

The approaches mentioned above do not consider event composition, that means that they filter events trying to deliver events of interest to consumers but without considering any correlation with other event occurrences. Event composition involves the occurrence of two or more primitive and/or composite events. Composite events are expressed using an event algebra, such as those defined in HiPAC [Dayal et al. 1988], Ode [Gehani et al. 1992], SAMOS [Gatziu and Dittrich 1993], and NAOS [Collet and Coupaye 1996]. Such algebras require an order function between events to apply event operators (e.g. sequence), or to consume events. To determine which of these events should be consumed/selected consumption modes[2] were defined [Charkravarthy et al. 1994]. Usually, events are timestamped to provide a time-based order with the purpose of facilitating event selection. But in open distributed environments global time is not applicable.

An approximation for modelling the time imprecision in distributed systems has been proposed. Assuming a sparse time base (where the points at which events can be generated are discretized and predetermined), Kopetz [1992] proposed the *2g-precedence* model. This model establishes that if events are at least two time granules apart, the sequence of these events can be determined unequivocally. Here an upper bound to the precision is assumed and a virtual clock granularity g is defined. Since the granularity depends on the assumed precision, it is not a feasible approach for wide area networks and open distributed systems.

Schwiderski [1996] adopted the 2g-precedence model to deal with distributed event ordering and composite event detection. She proposed a distributed event detector based on a global event tree and introduced a 2g-precedence-based sequence and concurrency operators. However, event consumption is non-deterministic in the case of concurrent or unrelated events. Additionally, the violation of the granularity condition (2g) may lead to the detection of spurious events.

[2] More details about consumption modes can be found in Appendix A – Section A.1.

The Cambridge Event Architecture (CEA) [Bacon et al. 1998] presents the *publish-register-notify* paradigm. Mediators provide the means to compose events. The implementation of CEA is based on a proprietary RPC system, limiting interoperability. Its successor, COBEA [Ma and Bacon 1998], extends the CORBA Event Service [Object Management Group 1997] with the CEA publish-register-notify paradigm, supporting fault tolerance, composite events, server-side filtering and access control. COBEA is also based on the 2g-precedence model.

In EVE [Geppert and Tombros 1998], an event-based middleware layer is proposed as a platform for a workflow enactment system. The workflow is mapped to services and brokers. The behavior of brokers is defined by ECA-rules using composition of distributed events. Typically, brokers are distributed over the network, therefore detection of composite distributed events is provided. Specifically, EVE requires chronicle consumption mode of events to correctly interpret workflow notifications. Notice that the chronicle consumption mode relies upon the temporal order of event occurrences consuming first the oldest occurrences out of the event stream.

In [Chakravarthy et al. 1999], the evolution of the Sentinel implementation [Liao 1997] is presented. Here they show a new architecture that abandons the initial implementation of a global event detector moving to a CORBA implementation. The problems of complex event detection in a distributed environment are not mentioned in this paper.

Yang and Chakravarthy [1999] present a formal refinement of Schwiderski's work extending the Snoop event algebra [Chakravarthy and Mishra 1994] to support event composition in distributed environments.

In CEDMOS [Baker et al. 1999], the implementation of a complex event detection and monitoring system for distributed environments is described. This project assumes that all clocks are generating timestamps that are synchronized but no details are given about how this is achieved.

Many projects on event composition in distributed environments such as [Bacon et al. 1998; Ma and Bacon 1998; Geppert and Tombros 1998; Collet et al. 1998; Yang and Chakravarthy 1999], either do not consider the possibility of partial event ordering or are based on the 2g-precedence model. Therefore, they suffer from one or more of the following drawbacks: they do not scale to open systems, they provide the possibility of spurious events, or they present ambiguous event consumption [Liebig et al. 1999].

Systems that support composite events must also address the semantic issues associated with processing composite events. For example, what must be defined is the manner in which timestamps are generated and the way in which events are selected and consumed.

Zimmer and Unland [1999] provide an overview of the semantic issues associated with composite events by presenting a formal meta model for studying, specifying, and comparing event languages. This meta model is based on three independent dimensions: a) an event instance pattern (which sequence of component events triggers a complex event), b) event instance selection (which event instances should be chosen as part of the complex event), and c) event instance consumption (which instances are consumed by a complex event). Using this meta model, they show that most of the existing event languages contain semantic inconsistencies and ambiguities.

In [Liebig et al. 1999] a new approach for timestamping events in large-scale, loosely coupled distributed systems is proposed. This uses accuracy intervals with reliable error bounds for timestamping events that reflect the inherent inaccuracy in time measurements. Additionally, this paper presents semantic ambiguities when using (centralized) event operators in a distributed environment.

2.4 Summary

This chapter presented related work in the three main areas that are associated with this thesis. The first section concentrated on data integration issues looking at approaches taken in federated multi-databases, event integration on active database systems and semantic integration on the Internet. Afterwards, different approaches to adapting centralized active databases to distributed environments were presented. Of particular interest is the unbundling approach that proposes the separation of the active functionality from the aDBMS and breaking it up into software units that can be rebundled later. In practice, the reuse of the essential component –the complex event detector– is not feasible since the inherent characteristics of distributed environments are simply ignored. Finally, two main issues related to distributed environments were revisited. The first one related to efficient event dissemination on such environments taking a look at academia projects and industry standards. The second included the difficulties associated with composite event detection in distributed environments presenting projects that have been working in this area.

Chapter 3

Foundation

This work is motivated by the requirements of the new generation of large-scale distributed applications. The goal here is to provide ECA-rule processing functionality with characteristics that are similar to those of a centralized aDBMS in a distributed component system to support the new generation of Internet-scale applications. The active functionality service proposed here is based on a flexible architecture founded on autonomous, combinable and possibly distributed services. The next section introduces the main ideas behind this service. After that, we show how to make rule specifications independent of the platform. Finally, all these issues are put together showing an integrated and clean approach.

3.1 Main Pillars

Ontologies play a fundamental role in this work; they are used integrally to deal with the integration of events and the interaction of autonomous services. In addition, because the architecture is based on components, ontologies are fundamental for the interaction among components developed independently. The underlying communication between these services is based on a publish/subscribe mechanism, which is suitable for distributed environments and offers other advantages as shown later in this section. In particular, this work emphasizes the following issues:

- a flexible architecture that can be adapted for different application scenarios,
- the use of an ontology to allow the integration of events coming from heterogeneous sources and also for the infrastructure itself,

- a platform for composition of events coming from heterogeneous sources in distributed environments that deals with partial orderings and the lack of a central clock, and
- the provision of an active service as a composition of other elementary services.

Three main pillars are the basis of this work: (1) an ontology-based infrastructure, (2) event notifications, and (3) a service-based architecture. In the next subsections these three aspects are presented. Built upon this foundation the service architecture proposed in this thesis is presented in Chapter 4.

3.1.1 Ontology-based Infrastructure

In our context, active functionality mechanisms are fed with events coming from heterogeneous sources. These events encapsulate data, which can only be properly interpreted when sufficient context information about its intended meaning is known. In general, this context information is left implicit and as a consequence, it is lost when data/events are exchanged across institutional or system boundaries. For this reason, to exchange and process events from independent participants in a semantically meaningful way, explicit information about its semantics in the form of additional metadata is required.

The architecture is founded on the use of shared concepts expressed through common vocabularies (ontologies) as a basis for interpretation of data and metadata. We represent events, or to be more precise event content, using a self-describing data model called MIX [Bornhövd 2000; Bornhövd and Buchmann 1999]. In the following, we refer to events represented based on MIX, i.e. based on concepts from the common ontology as *semantic events*. MIX refers to concepts from a domain-specific ontology to enable semantically correct interpretation of events, and it supports an explicit description of the underlying interpretation context. Simple attributes of an event (also known as simple semantic objects) are represented as triplets of the form $< C,\ v,\ \$ >$, with C referring to a concept from the common ontology, v representing the actual data value, and $\$$ providing additional metadata (also known as semantic context) to make implicit modeling assumptions explicit. This semantic context specifies the interpretation context of a data value and is also represented with MIX concepts. For example a bid amount can be represented as <BidAmount, 99, {<Currency,"USD">, <Scale,1>,...}> .

Complex semantic objects are represented in the form $CSO = < C,\ \mathbb{A} >$, where C refers to a concept of the ontology, and $\mathbb{A}$ provides the set of simple or complex objects

representing its sub-objects (also known as attributes). For example, a PlaceBid event can be represented as follows:

CSO = <PlaceBid, { <ParticipantId, 412, {<IdentifierCode,"eBayCode">}>,
<ItemId, 5423, {<IdentifierCode,"eBayCode">}>,
<BidAmount, 99, {< Currency, "USD" >, < Scale, 1 >,...}>,
...} >

Semantic events from different sources can be integrated by converting them to a common semantic context using conversion functions. Conversion functions can be specified in the underlying ontology if they are domain-specific and application-independent. Application-specific conversion functions may be defined and stored in an application-specific conversion library [Bornhövd and Buchmann 1999].

As depicted in Figure 3.1, ontologies are used in this thesis at three different levels: a) the basic level, where elementary ontology functionality and physical representation is defined; b) the infrastructure level, where basically concepts of the active functionality domain are specified; and c) the domain-specific level, where concepts of the subject domain (e.g. online auctions) are defined.

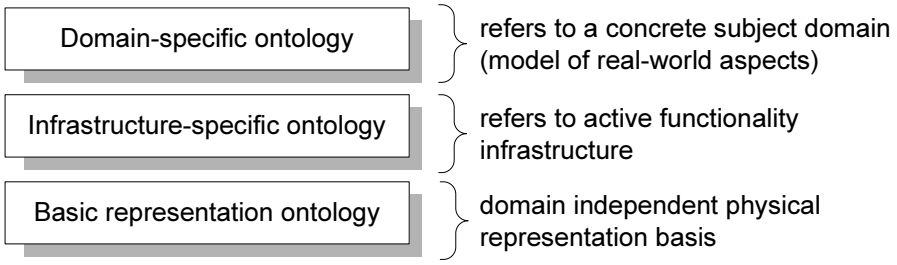

Figure 3.1: Three categories of ontology concepts

Basic representation ontology: Here the physical representation of data/concepts (e.g. number, string) for the higher levels of the ontology are defined. It is domain-independent.

Infrastructure-specific ontology: All elements related to active functionality are represented by concepts defined here. Difficulties associated with different rule language dialects, ambiguities and imprecise terms are resolved using an explicit common vocabulary. For instance, the terminology related to the definition of rules such as Event, Condition, Action, ECA-Rule, and so on, are explicitly defined in the infrastructure-specific ontology. Furthermore, other aspects related with the infrastructure, for instance, notification-related terms are also captured in the infrastructure-specific ontology.

Domain-specific ontology: Real-world concepts like, PlaceBid or StartOfAuction are defined in the corresponding domain-specific ontology (e.g. Online Auctions). A phys-

ical representation is associated with a domain-specific concept definition by inheriting from a concept of the basic representation.

The separation of infrastructure- and domain-specific ontologies is clearly shown in Figure 3.3.

3.1.2 Events and Notifications

An *event* is understood here as a happening of interest. Events are classified in this work as follows:

- *Database events* that are further subdivided into data modification and data retrieval events. Data modification events correspond to those operations that alter data in the database (like SQL operations insert, delete and update in relational databases). Data retrieval events are, for instance, the execution of a selection in a particular table in a relational database, the fetch of an object, or the invocation of a particular method that retrieves objects in an object-oriented DBMS. Other internal events can also be considered, but this depends on the support provided by the database engine to detect these situations.

- *Transaction events* refer to the different stages of transaction execution, e.g. begin transaction, commit, rollback, etc.

- *Temporal events* are classified into absolute, periodic and relative. Absolute temporal events are defined using a particular day and time, e.g. StartOfAuction as "January 29, 2001 at 10:00AM", while periodic temporal events are signaled repeatedly using time or calendar functions, e.g. every "Friday at 11:59 PM". Relative ones are defined using a time period with respect to another event, e.g. one week after StartOfAuction.

- *Abstract events* or application-defined events are declared by an application denoting an event, e.g. UserLogin, AuctionCancelled. Events of this kind are signaled explicitly by the application. In other cases, where event sources do not explicitly announce happenings, some events can be observed. For instance, in platforms like CORBA and J2EE, service requests can be intercepted. Using this feature, happenings related to a method execution can be intercepted transparently (without modifying the application). There are also event sources where no interception is possible. In those cases, event sources need to be polled in order to detect these events.

This event classification presented above is presented in Figure 3.2 with the help of a UML class diagram. This classification is explicitly specified in the infrastructure-specific ontology. Notice that ontologies in the scope of this work can evolve making them extensible, for instance extending/specializing this classification. For example, a heartbeat[1] can be specialized from the periodic temporal event, adding, in this case, supplementary attributes, such as frequency, process identification, etc. Likewise, real-world aspects of a particular domain are represented at the domain-specific layer, e.g. StartOfAuction as a specialization of an absolute temporal event; EndOfAuction as a specialization of a relative event; NewAuctionParticipant as a creation of a new tuple in the participant's data representation; ParticipantLogin as application-defined, and so on.

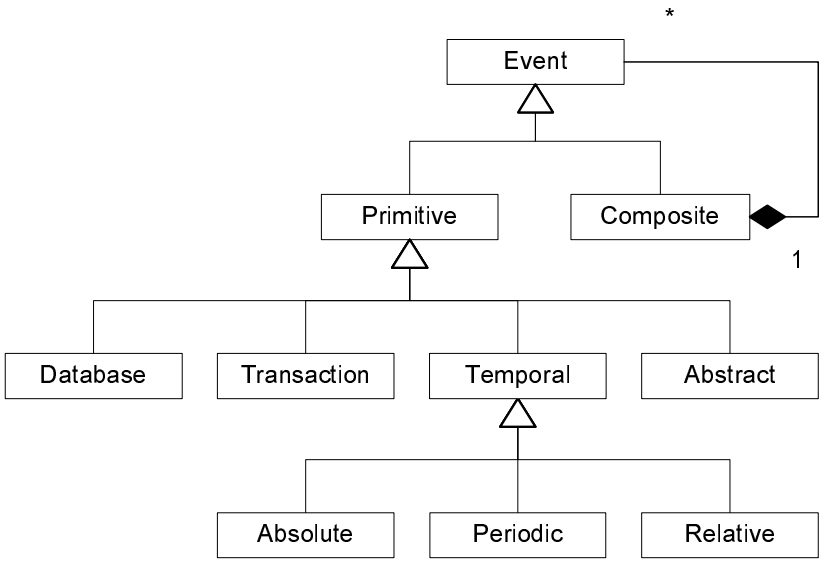

Figure 3.2: Event classification

Events coming from diverse sources must be mapped to the common vocabulary. This is basically the task of *event adapters*. They convert source-specific events into their corresponding concepts of the ontology augmented with semantic contexts. The association of context information with events serves as an explicit specification of the implicit assumptions about the meaning taken inside the event source. Without this additional information the event content cannot be correctly interpreted once the event leaves the source boundaries. However, incoming semantic events representing the same concept may still be dependent on the respective source, represented with different semantic contexts, i.e. on the basis of different units of measure, coding conventions, etc.

[1]The heartbeat protocol is based on a message sent between machines at a regular interval with the purpose of monitoring the availability of a resource.

Therefore, based on the explicit description of the underlying context these semantic heterogeneities can be resolved by converting the data to a common context using appropriate conversion functions. This common context is specified by the consumer of the event. For instance, consider the placement of a bid that is generated at an american auction site. This happening is then mapped into the PlaceBid concept (that is defined in the ontology) and the assumptions about the data involved are attached in the form of semantic context. Taking a closer look at one of its attributes, e.g. the bid amount, it is augmented with the currency in question in order to be correctly interpreted outside this auction house.

A *notification* is a message reporting an event to interested consumers. A notification carries not only an event instance but also important operational data, such as reception time, detection time, event source, time-to-live, etc. As has been seen in several active system prototypes, complex event detection is mainly based on operational data (particularly correlating timestamps of event instances) while filters are based on both (i.e. events can be discarded by comparing attribute values of the event content or for instance, by looking at the event source or time-to-live). For this reason, here the content of a notification distinguishes between operational data and the event content. This distinction facilitates the implementation of complex event detectors and filters. Concepts related to notifications and in particular to operational data (e.g. Notification, OperationalData, DetectionTime, EventSource, TimeToLive) are specified as part of the infrastructure-specific ontology. On the other side, concepts related to the event content should be specified on the corresponding domain-specific ontology. Figure 3.3 depicts this organization, where ellipses symbolize ontology concepts.

A *notification service* based on a publish/subscribe paradigm[2] is responsible for delivering events to interested consumers. Here a notification flows from an event producer possibly to a set of consumers. Subscribers (consumers) place a standing request for events by subscribing. As well as this, a publisher makes information available for its subscribers. A publish/subscribe mechanism provides asynchronous communications, it naturally decouples producers and consumers, it makes them anonymous to each other, it allows a dynamic number of publishers and subscribers, and it provides location transparency without requiring a name service. The notification service uses *concept-based addressing* in order to provide a higher and common level of abstraction to describe the interests of publishers and subscribers.

[2] An overview of the concepts of publish/subscribe communications can be found in Appendix A.

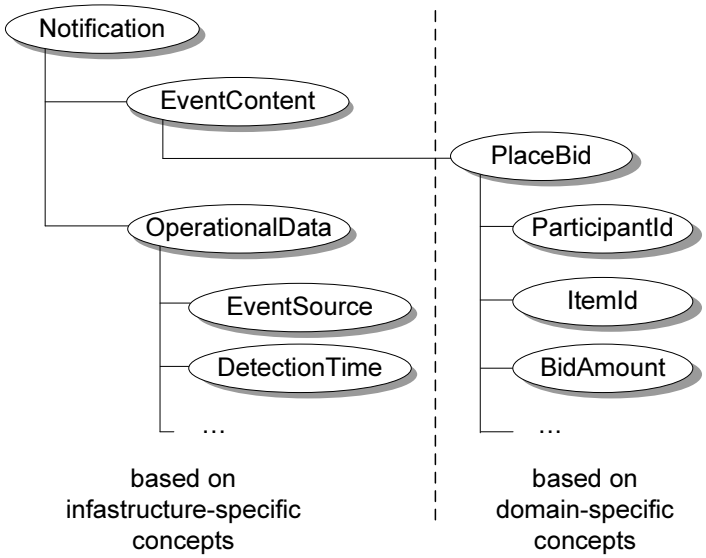

Figure 3.3: Schematic view of a PlaceBid notification

3.1.3 Service-based ECA-rule Processing

In this work, traditional ECA-rule processing is decomposed into its elementary parts (called here elementary services). These autonomous services are responsible for complex event detection, condition evaluation, and action execution. Elementary services expose two kinds of generic and very simple interfaces: i) a *service interface* with a single method that receives an event notification as a parameter; ii) a *configuration interface* that is used for administration purposes, such as registration, activation, deactivation, deletion, etc. This simple service interface provides flexibility, enabling a simple collaboration/interaction among services. ECA-rule processing is then realized as a combination/composition of these elementary services according to the rule definition. From an abstract point of view, this service composition takes the form of a chain of services, where event instances flow through the composed services in order to carry out the corresponding rule processing. Interactions among elementary services involved in the processing of a rule are based on the notification service.

But before processing rules, services must be configured for this purpose. This is the task of the *ECA-rule Manager* which plays the role of a representative of the active functionality service offering operations needed to define, remove, activate, and deactivate ECA-rules. This means that administration activities are executed through this representative.

The most complex activity is the registration of a rule, which involves the composition of elementary services that participate in its processing. This composition consists

basically of four steps: i) decomposing the rule, ii) finding, iii) contacting, and iv) configuring elementary services. The ECA-rule manager decomposes the rule definition passed for registration and based on its parts it should find adequate elementary services in the service registry. At this point, the ECA-rule manager is responsible for building a chain of elementary service that will be in charge of processing the rule in question. Afterwards, elementary services are deployed (if needed) and contacted for configuration. The configuration of an elementary service itself comprises the subscription to the output of the preceding elementary service, the configuration of the task under the responsibility of this service (e.g. a condition evaluation service is configured with the condition of the rule that must be evaluated) and the configuration of the publisher.

As mentioned before, interactions among elementary services rely on the notification service. Coupling modes (that in this case specify the transactional relationship between elementary services involved) can be delegated to a notification service that supports them. For instance, the notification service proposed in the X^2TS Project [Liebig et al. 2000b] integrates notifications and transactions allowing the specification of coupling modes to be made on a per subscription basis [Liebig et al. 2000a; Liebig and Tai 2001].

For instance, consider the registration of a rule R1 with the ECA-rule manager (1), as shown in Figure 3.4 (where boxes denote services, lollipops for their interfaces and circles inside these boxes represent instances of objects under the control of the corresponding service). The input for the registration is a rule description represented using the ontology (as it will be explained later in this chapter). The manager breaks the incoming rule into elementary pieces (2) and searches for proper services according to the pieces obtained also considering some other configuration factors (3). Notice that these tasks can be influenced by a system administrator. Afterwards, the manager registers events corresponding to R1 with the services obtained (4a, 4b and 4c). The complex event detector configures internal objects in order to detect R1's event; then the condition evaluation service instantiates a condition object and subscribes it with R1's event object. Next, the action execution service instantiates an action object and subscribes it with R1's condition object, completing the *service composition phase*.

At run-time, when the complex event of R1 is detected, the complex event detector publishes this happening. This means that all rules that were defined using this triggering event are automatically "fired", in particular, their condition objects are notified (step 5a). In this situation, no conflict resolution policy is needed because all rules are executed concurrently (other execution models are possible). When condition objects are notified, they evaluate their predicate and, if true, they automatically notify the corresponding action objects using the same notification service (step 5b).

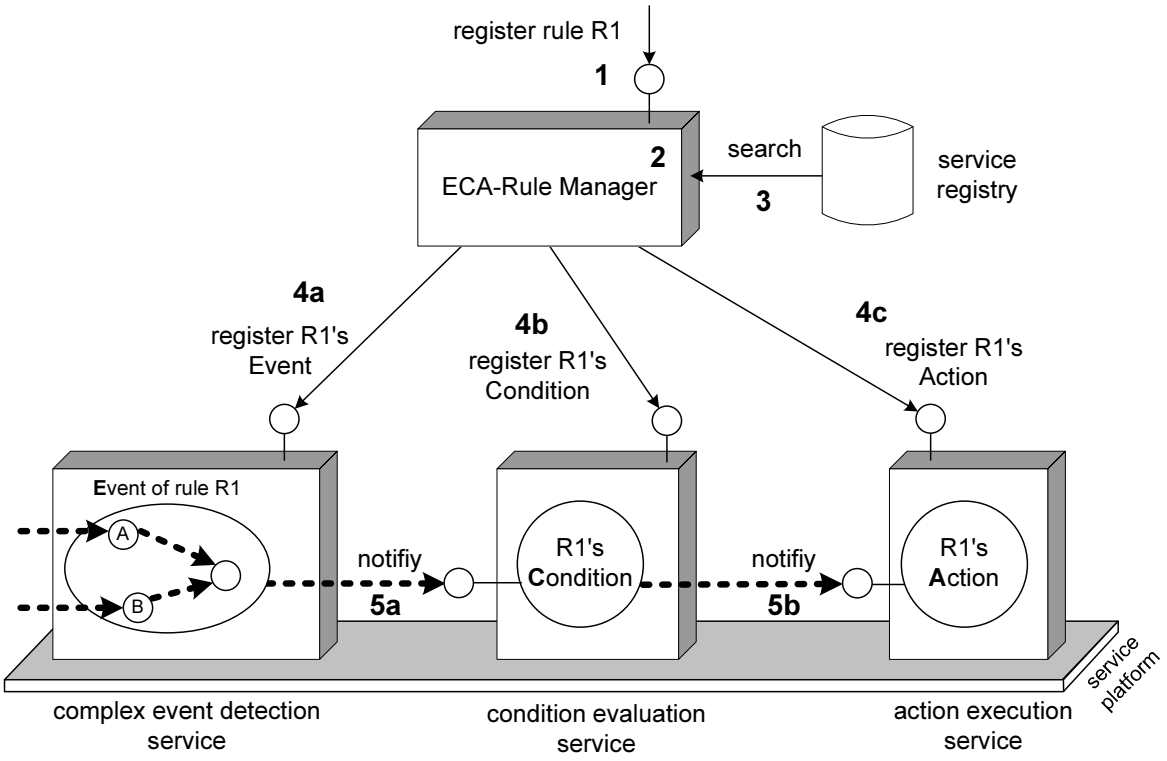

Figure 3.4: Interaction among elementary services (ECA-rule processing chain)

Interaction among services could be done using other mechanisms than publish/subscribe, e.g. remote procedure call. However, the publish/subscribe paradigm plays an important role in our architecture providing the following advantages:

- it allows asynchronous communication and decouples event producers and consumers (suitable for open distributed environments),
- it is particularly useful if various rules are associated with the same event,
- it facilitates concurrent rule execution (no centralized rule selection mechanism is needed),
- the notification contains required event information and its context (context propagation),
- it provides a simple and powerful generic communication model, and
- it helps to represent dependencies of the flow of work explicitly.

3.2 Defining Rules

Regarding rules different perspectives need to be distinguished: how business rules are expressed by the users and how they are represented inside the system. Taking this into account, in this work rule representation is organized into three layers:

- *external:* it allows the possibility to tailor a rule definition for each specific domain (or group of end-users) making the specification of rules convenient without the complications or levels of detail imposed by a generic rule definition language.
- *conceptual:* it provides independence between the implementation of the underlying active mechanism and an end-user's rule definition.
- *internal:* it enables the use of a "generic" active functionality service where components or services that are involved can be implemented using different optimization criteria or different programming languages, but they all "understand" the conceptual layer and they use a common internal representation to process rules.

Associated to each of these layers are users: i) those who define the ECA-rules, known as *end-users*, and ii) those who provide end-users the means to define their rules, called *system developers*, and iii) those who implement the active functionality service, known as *service developers*. Figure 3.5 illustrates the rule representation layers and their corresponding users.

It must be borne in mind that domain-specific and infrastructure-specific terminology are represented here using ontologies as described previously. On this basis, developers can provide various "external" alternatives to end-users in order to define rules taking into account the domain in question, the target end-users, etc. One alternative is just to create a rule specification language according to a particular domain. By means of a textual description that conforms to the language, end-users can define ECA-rules. This rule description is then compiled, validated and transformed into a conceptual representation of rules. Another alternative consists of the use of a user interface (like a form) where end-users fill out fields or choose valid options from a menu. For instance, in order to specify the event that fires a rule, it can be selected from the set of possible events. Once the form is submitted, values from all fields are validated and they compose the corresponding conceptual rule representation.

Other alternatives can be explored by developers to provide end-users with appropriate ways to define rules. But from the developer's point of view, all these alternatives rely on an *Ontology API* that facilitates the access and manipulation of the ontologies. In

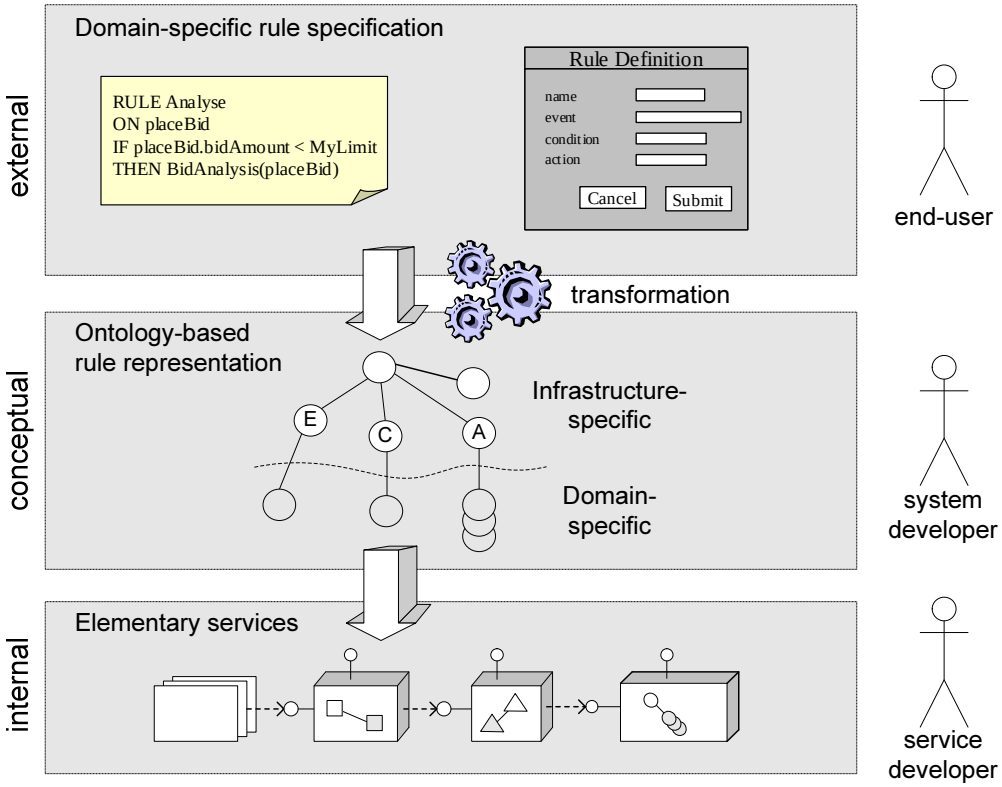

Figure 3.5: Rule representation layers

this way, all kinds of external rule definitions produce an ontology-based (conceptual) rule representation as output.

Figure 3.6 shows a meta-definition of the conceptual rule representation. Concepts of the ontology are depicted using ellipses. Those with gray background represent mandatory concepts (also known attributes).

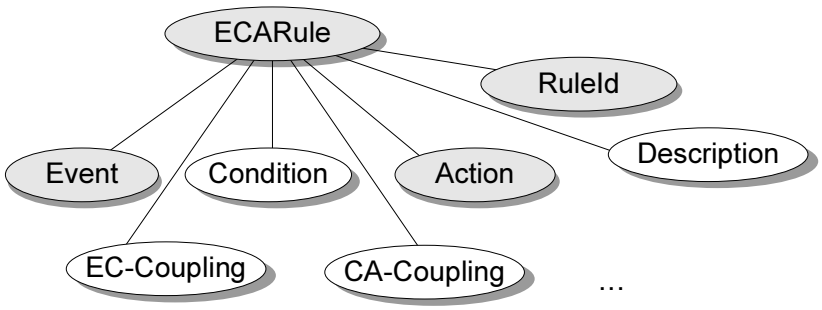

Figure 3.6: Meta-definition of the conceptual rule representation

As mentioned above, this conceptual rule representation provides independence between the underlying active mechanism and the end-users' rule definitions. This permits system developers to make available to end-users the most appropriate way to define rules for a particular domain without restricting/complicating the definition due to generic aspects.

Additionally, this independence enables the use of a "generic" active functionality service for different domains, making the underlying service independent of the rule specification. Elementary services can be implemented by independent system-developers using different programming languages but they all receive a common conceptual description as input, which can be then transformed and represented internally with other means.

With the aid of an integral use of ontologies as part of the infrastructure, the definition of rules can benefit from the use of semantic contexts. Contexts can be associated to conditions and actions in order to evaluate them under the defined contextual information. For instance, a condition predicate that verifies distances can define "metric system"as context. In this manner, incoming events from heterogeneous sources are first converted into the metric system (if necessary) before they are used for evaluation. Consequently, conditions and actions are always specified at a domain-specific level, and are independent of source-specific representations. This provides a very useful and powerful mechanism for interpreting events from heterogeneous sources by maintaining a high-level specification.

3.3 The Big Picture

The use of the active functionality service presented in this work involves basically three main phases (as illustrated in Figure 3.7): rule definition, service composition, and rule execution.

The *rule definition* involves the specification of ECA-rules by end-users. These rule definitions are then processed, analyzed and transformed into a rule description based on the ontology (conceptual representation).

After defining rules, what follows is the *service composition*. Here, the rule description for a rule obtained from the previous phase is registered with the ECA-Rule Manager (the representative of the active functionality service). The registration activity is responsible for composing all required elementary services involved in the execution of rule in question. Conceptually, this service composition takes the form of a chain of processing services.

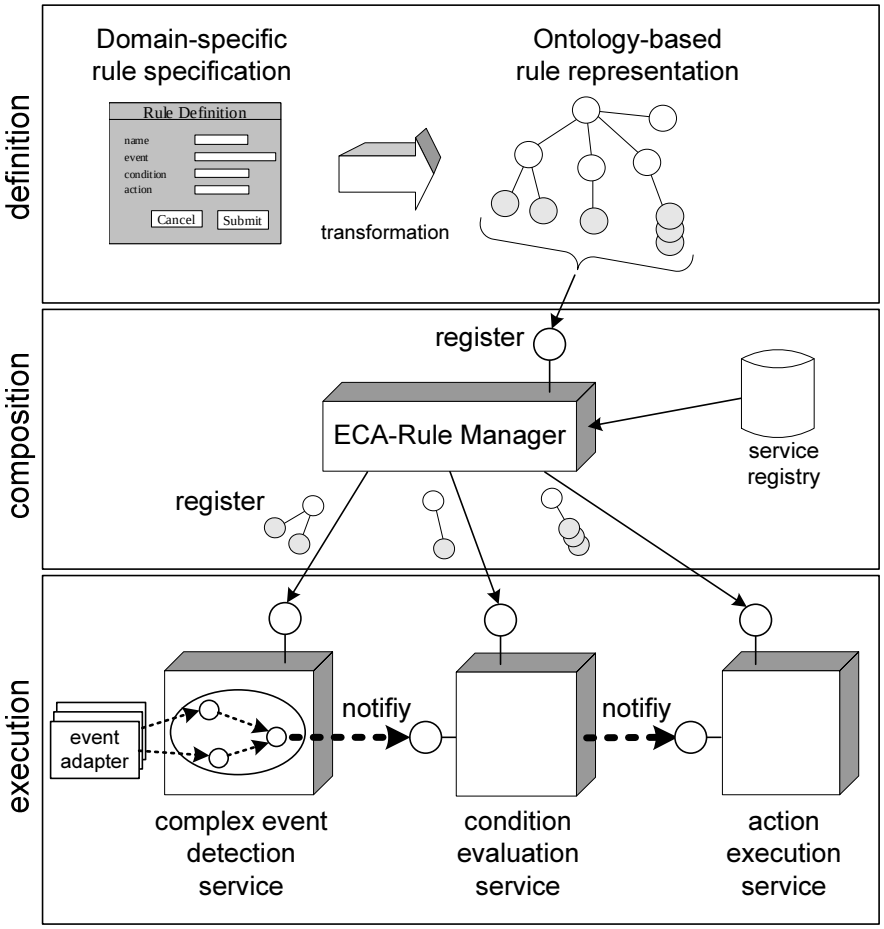

Figure 3.7: The big picture

Once required services are configured, it is the turn of the *rule execution* phase. Involved services wait until events (sent as notifications) are signaled. Notice that events produced at event sources are then transformed into semantic event with the help of event adapters. When a notification arrives at an elementary service, it is processed and if appropriate, it is republished in order to turn over to the next service in the processing chain.

3.4 Summary

Ontologies form part of the foundation of the proposed active functionality service and they are employed in an integral way. They are used as a common interpretation basis

to enable semantically correct interpretation, in this case relying on the MIX model. The ontology approach is not only applied to integrate events from diverse sources but also to support the interaction among elementary services at semantic level, to address event notifications, and to represent ECA-Rules. In this work, ontologies are organized in three layers (basic representation, infrastructure-specific, and domain-specific) with the purpose to clearly separate the infrastructure from the terminology related to the problem that is being solved.

Events are disseminated with the help of a publish/subscribe notification service which is based on concept-based addressing. This addressing approach provides a higher and common level of abstraction to describe the interests of publishers and subscribers.

The ECA-rule processing was decomposed into elementary services that provide two very simple and generic interfaces. The rule processing is effectively materialized as a composition of these elementary services according to the rule definition. The resulting composition forms a chain of services that are in charge of processing a particular rule. The composition of these very simple elementary services provides flexibility by allowing a simple way to configure the flow of services that participate in the execution of rules. The interaction among elementary services is accomplished by means of the notification service mentioned above.

The conceptual rule representation provides independence between the underlying active mechanism and the end-users' rule definitions. This permits system developers to tailor the most appropriate way to define rules for a particular domain without restricting/complicating the definition due to generic aspects. Moreover, the definition of rules can include contextual information. This enables a higher level of abstraction without taking care of source-specific event representation peculiarities while making possible the correct interpretation of data involved in events, conditions and actions.

Chapter 4

Service-based Architecture

This chapter contains a description of the proposed active functionality service architecture and a brief explanation of the elementary services involved. Additionally, the role of event adapters is explained. Finally, we formalize the proposed architecture.

4.1 Framework

The main idea of the approach taken here is simply to have an active functionality service that can be configurable according to different scenarios while solely deploying the functionality that is necessary according to the set of rules defined.

ECA-rules to be processed are decomposed into their elementary parts, e.g. event, condition, action. Moreover, events can be primitive or composite, and condition may be restricted to predicates on event attributes or may involve external systems. According to this, to each elementary part of a rule corresponds an elementary service that can process it. Consequently, elementary services are restricted to fulfill only one task, e.g. condition evaluation, action execution. There could be more than one implementation of an elementary service.

In contrast to building complex peer-to-peer wiring among components, here all elementary components have a uniform, simple, and well-defined (run-time) interface. Additionally, arguments involved in the interface definition are defined using concepts of the infrastructure-specific ontology. Elementary services are implemented using loosely-coupled components to fit distribution requirements and they can be combined without requiring changes in their code. This is achieved by using notifications that flow from one service to the other. The destination of the outgoing notifications relies

basically on a publish/subscribe mechanism, which is used for notification dissemination.

To carry out a major task processing elementary services must be composed. For instance, to process a rule, selected elementary services must be combined according to the definition of the rule in question. As a result and from an abstract perspective, this combination forms a rule processing chain. Figure 4.1 shows a combination of services to support typical active functionality.

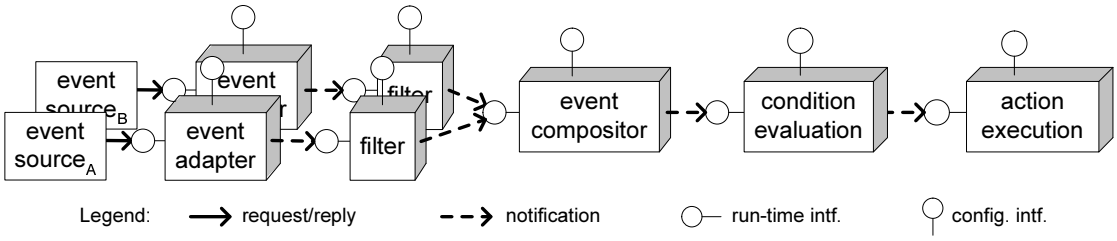

Figure 4.1: Elementary components to support active functionality

Notice that the rule processing chain always begins with the event source (and its corresponding event adapter) and ends with the action execution service. In-between, optional elementary services, such as event filters or condition evaluation, can be interconnected. For instance, consider an EA-rule (where the condition is omitted) and assume that the corresponding event is a complex one. To this particular rule corresponds the service configuration depicted in Figure 4.2 (a). Here the flow of processing begins with the event adapter which is connected to the event compositor (or complex event detector) which in turn is connected with the action execution service. Another service configuration is illustrated in Figure 4.2 (b), where an ECA-rule with a simple event and a condition predicate is assumed.

As shown in the previous figures, elementary services have a configuration interface. By means of this interface, elementary services are configured with the pattern needed for subscribing the notifications of interest, with the part of the rule that must be processed and, if needed, with the information to publish its output.

With the purpose of facilitating the use of concepts in software development an Ontology API was provided. Additionally, an ontology service is used to manage concept specifications defined at the three layers of the ontology-based infrastructure and, to provide access to the implementation of ontology concepts.

Information about rule definitions, deployment and configuration of their events, conditions and actions, and additional information about services, must be maintained in a repository to support configuration and maintenance.

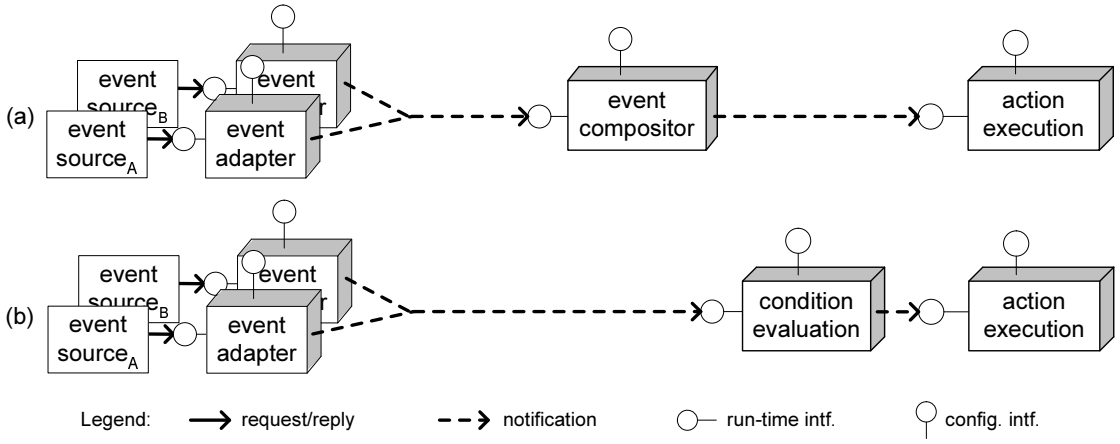

Figure 4.2: Different elementary service configurations

This architecture is designed to run on top of a service platform that facilitates the deployment and management of the services mentioned. Details can be found in Section 6.2.

4.2 Event Adapters

As mentioned in the previous section, event adapters are the first element in the rule processing chain. They are responsible for converting source-specific events into semantic events (those reflected in the ontology with their corresponding semantic context attached).

For instance, consider the scenario presented in Figure 4.3 where an application signals the placement of a bid through its adapter. The adapter maps application-specific representations into the corresponding terms defined in the ontology. Moreover, context information (such as currency, date format, coding information, etc.) is attached to shape a semantic event. This semantic event is shown in the picture using a textual representation.

As can be seen in the figure the assumptions inside the application are left implicit. However, this kind of information is required when exporting data/events outside the boundaries of this particular application. Making these assumptions explicit (at the event adapter) events produced at this application (or event source) can be correctly interpreted by diverse consumers.

A list of examples of event adapters are illustrated at the bottom of Figure 4.4 and briefly described below:

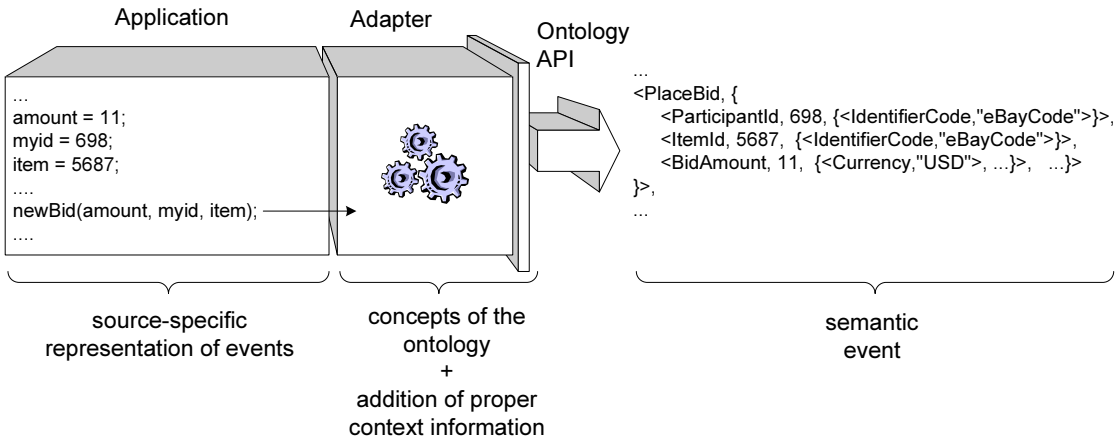

Figure 4.3: Functional view of an adapter

- *XML adapters* translate XML-based messages into semantic events.
- *Database adapters* are used to signal database operations outside the database in the form of semantic events. With this purpose, trigger mechanisms can be used to detect an event and then stored procedures are used to generate the corresponding semantic event.
- *Interceptors* are used to intercept a service request (before or after request processing). Once a request is intercepted, a corresponding event can be signaled.
- *E-mail adapters* intercept e-mails according to specified e-mail properties, like sender address, subject, etc. Once received, the necessary information is extracted from a (semi)structured e-mail and converted into a semantic event.

4.3 Services Involved

This section describes all elementary services that could be involved when processing ECA-rules. Notice that probably not all of these services are required for the execution of a rule. Depending on rule definition, services are selected and configured by the ECA-Rule Manager to collaborate with its processing. Implementation details of these services can be found in Chapter 6. Figure 4.4 shows a global overview of elementary services and event adapters.

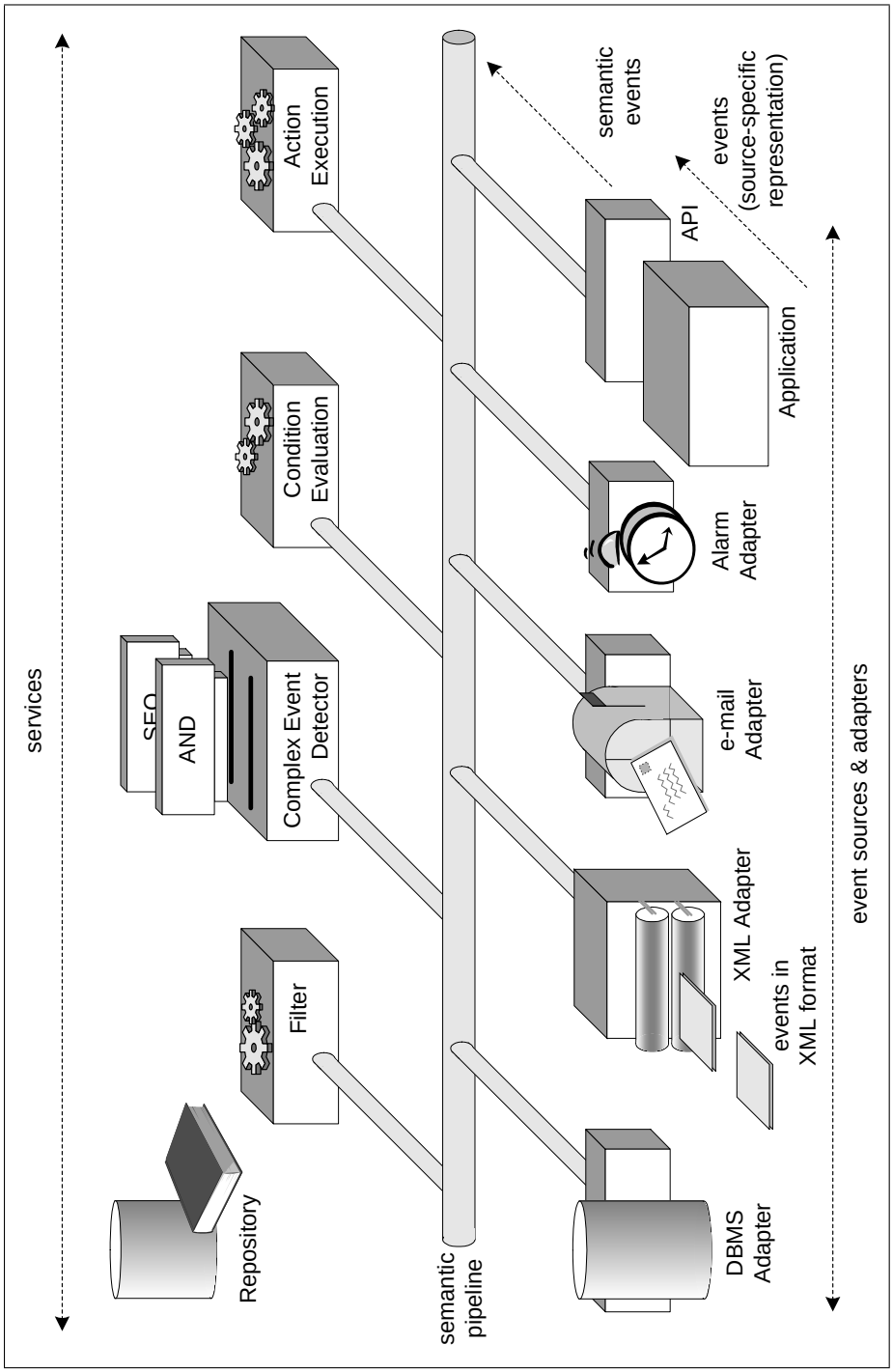

Figure 4.4: Adapters and elementary services

4.3.1 Notification Service

In this work, a notification carries an event occurrence from its producer to possibly a set of consumers. A notification service is responsible for delivering notification to interested consumers. In this particular case, the publish/subscribe paradigm was selected because it fits very well in the proposed approach (as was mentioned in the previous chapter).

To target notifications a *concept-based addressing* approach is proposed in this thesis. As its name suggests, subscriptions are made based on the concepts defined in the underlying ontology providing a higher and common level of abstraction to describe the interests of publishers and subscribers. On this basis, consumers do not need to take care of proprietary representations. Moreover, if new concepts are defined, nothing need to be changed at the consumer's side.

This approach is based on the subject-based addressing principles where the subject name space is hierarchically organized. Here concepts are mapped to the subject name space where the concept name and some of its attributes form part of the subject.

The notification service is in charge of communications between all the elementary services involved. This situation is schematically depicted by a pipeline in Figure 4.4.

4.3.2 Alarm Service

This service is also considered as a source of temporal events (absolute, relative and periodic). These events require a clock scheduler in order to signal scheduled temporal events. For instance, an absolute temporal event indicating StartOfAuction, can be defined as "March 23, 2002 at 9:00AM GMT+1" which means that at the specified time this event must be signaled.

This service can also be useful for the infrastructure itself. For instance, in distributed systems a heartbeat mechanism is sometimes needed for producing (and consuming) heartbeats within a determined periodicity.

4.3.3 Timestamp Service

Timestamps are usually used to allow events to be ordered. This order takes a fundamental role in event consumption (e.g. chronicle) and in event composition (e.g. time-related event operators like sequence). Depending on the given system environment, e.g. distributed or centralized, different timestamp models can be used. For

example, for centralized environments, a simple timestamp mechanism may be sufficient since only one clock is used to timestamp events. For distributed closed networks, the 2g-precedence [Kopetz 1992] model may fit. For open distributed environments, the accuracy interval model [Liebig et al. 1999] can be employed. Notice that in distributed environments clocks of computers involved must be synchronized.

The interesting point of this architecture is that the most appropriate timestamp model can be used according to the characteristics of the environment in question. The implementation of timestamp services should explicitly declare all assumptions made. For this reason, a timestamp ontology is needed to describe all terms related to the timestamp mechanism in question, like, observation (detection time, occurrence time, reception time), clock source (local, local synchronized), clock granularity, etc.

4.3.4 Complex Event Detection Service

This service is responsible for detecting complex situations based on primitive and composite events. These situations of interest are described using event operators.

The approach proposed in this thesis follows the idea of components and containers. Components are the event operators that are plugged into the container (see top center of Figure 4.4). The container itself is the complex event detector kernel which controls the event detection process. Event operators (which basically contain the operator's logic) are simply plugged into the container in order to detect the situation of interest. Details about this approach are given in Chapter 5.

4.3.5 Condition and Filter Services

Conditions and filters are two similar ways of defining a boolean predicate on events. Filters are used to select notifications by discarding those that do not verify the predicate. Conditions, on the other hand, are used as a precondition before the rule's action is executed. In both cases, these predicates can include an evaluation context as explained in Section 3.2. The next subsections describe these services.

4.3.5.1 Filters

Filters discard notifications based on predicates trying to reduce network traffic and minimizing the amount of events considered for firing rules. These predicates involve only attributes contained within a notification (intra-notification). That means those

attributes that are part of the notified event itself and those that form part of the operational data. For instance, a predicate can express that notifications coming from a particular event source must be discarded or for example, that bid placements (PlaceBid) over EUR 100 must be discarded. This can be expressed as follows:

context: { <Currency, EUR>, ...}
predicate: PalceBid.BidAmount $\leq$ 100

Here the use of the context information allows an automatic conversion (if needed) of all incoming bid placements coming from diverse heterogeneous sources.

A filter service is in charge of evaluating defined predicates on event instances. There are three different alternatives with respect to the location of filters. The first one, as near as possible to the event source, delivering only events of interest reducing network traffic. The second is on the consumer's side, so events are delivered to the consumer and are then filtered. Finally, the filter can be located as a mediator between producers and consumers. Here events are filtered at the mediator and only those of interest are re-transmitted to the consumer. This last alternative seems to be of special interest for event consumers in a mobile environment which may connect sporadically or are connected to the network over a low bandwidth channel.

4.3.5.2 Condition Evaluation

Conditions are boolean predicates with the purpose of ensuring some particular state once the rule is fired and before its action is executed. A condition is expressed as a boolean predicate and it can involve (boolean) functions and attributes from notifications (including the event itself and a notification's operational data). For instance, a condition that expresses that a particular auction item is restricted to be auctioned among participants in Germany can be expressed as follows:

PlaceBid.ItemId == 1234 AND database.livesInGermany(PlaceBid.ParticipantId)

In this case, the predicate verifies whether the incoming PlaceBid event corresponds to the item in question, and taking the participant's identification (ParticipantId) as a parameter it verifies through a function if she lives in Germany.

At configuration time this service sets its pertinent properties (predicate to evaluate, plug-ins that allow the interaction with external systems, etc.), and subscribes to its corresponding triggering event by taking into account the specified coupling mode between event and condition. When an incoming notification arrives, the condition is evaluated and if true, the notification is republished in order to reach the next service in the processing chain.

Notice that conditions differentiate from filters in that a filter's predicates are restricted to refer to attributes of the event (including also those of the notification as well) whereas conditions can also involve functions on external systems.

4.3.6 Action Service

An action is a command, statement or a sequence of them, which can include event attributes as parameters. In the same way that an evaluation context can be attached to filters and conditions, contexts can also be associated with actions in order to transform/convert value attributes to the appropriate context. For instance, consider an action of a rule that appends all bid placements in a database table (in order to maintain the history of all auctions). Because bid placements can be expressed in local currency, bid amounts need to be converted to a common representation (e.g. Euros) before they are stored. This can be expressed as follows:

```
context: { <Currency, EUR>, ...}
action: insert into BidHistory
        values (PlaceBid.ParticipantId, PlaceBid.ItemId, PalceBid.BidAmount,...)
```

In order to access external services or systems (e.g. access databases, invoke methods on distributed objects or wrapped legacy applications, invoke transaction control operations, etc.), condition and action services use plug-ins. Plug-ins are responsible for delegating the execution of instructions to the system they represent. Moreover, they are in charge of maintaining the target context of the system they interact with making possible the automatic conversion of data to the target system. Details about plug-ins can be found in Section 6.7.

As well as other services, the action service provides an interface to register a rule action which involves the setting of properties and the subscription to the corresponding event of interest. Once notifications are received, they are bound to the action statement which is executed according to the coupling mode specifications.

4.3.7 Repository and Ontology Service

Here an ontology server is used to store, manage and provide access to the concepts of the underlying ontolgies. This vocabulary provides the domain- and infrastructure-specific description basis to which all other services refer. The ontology server provides a common access point to the vocabulary, and it provides concept definitions and textual, human-readable descriptions of available concepts for interactive exploration by developers and end-users.

In the repository are stored: rule definitions, their deployment and configuration, event constellations, configuration of adapters, service configurations, and the subject namespace organization. This repository is used for configuration and maintenance purposes.

4.4 Formalization

In this section, the semantics of components is formalized. Components are abstract devices which have initialization, input and output interfaces.

Components can be assembled into more complex components. Output interfaces are always attached to input interfaces, and only certain component combinations make sense. Interfaces in this framework are extremely simple and can be thought as a single method with a single argument.

Component behavior is specified according to Hoare Logic [Lamport 1980]. That is, for every component $\mathcal{Z}$ we give a triple $\{P\}\mathcal{Z}\{Q\}$ where P and Q represent the pre- and postconditions of $\mathcal{Z}$ respectively. Pre- and postconditions are predicates on the state of the interface. The triple $\{P\}\mathcal{Z}\{Q\}$ means that if P holds on the input interface, then component $\mathcal{Z}$ will terminate and after termination, predicate Q will hold on the state of the output interface.

4.4.1 Notification Service

Components interact with other components via notifications. Notifications here are abstract. Two special components make this possible, by encapsulating the interaction mechanism. The first one, the *subscriber*, is used to receive notifications from other components and the second one, the *publisher* is used to send notifications to other components. These two components can be seen as the glue needed to combine components and at the same time they are the interface to the underlying communication mechanism. In this particular case, a publish/subscribe approach is defined for component interaction. It should be noticed that the underlying communication mechanism can be replaced if necessary.

Subscribers and publishers are attached to the input and output interface of a component respectively.

The rest of this section presents the formal definition of notifications, together with the definition of subscribers and publishers. These definitions are then helpful to define other services.

Notifications

Assume that $\mathcal{N}$ is the set of all *notifications* including the empty notification ϵ.

$$\mathcal{N} = \{n_1, n_2, ...\} \cup \{\epsilon\}$$

A tag is associated with each notification that is processed by the underlying notification service. The tag consists of a general ontology-related part and an application-specific part. These are formalized as predicates, i.e. boolean function over notifications.

Let $\mathcal{B}$ be the set of all boolean functions over $\mathcal{N}$. A *tag* ω is an element of $\mathcal{B} \times \mathcal{B}$, i.e. a tuple (a, c) where a represents the application-specific part of the tag and c represents the ontology-related part of the tag. For example, $c_{PlaceBid}$(n) is true if and only if n is an instance of a PlaceBid notification. The application-specific tag is used for routing purposes, e.g. to ensure that notifications flow through the semantic pipeline correctly.

Let $\Omega = \{\omega_1, \omega_2, ...\} \subseteq \mathcal{B} \times \mathcal{B}$ denote the set of all possible tags in the application context, where $\omega = (a, c)$ such that

$$a : \mathcal{N} \longrightarrow \{T, F\}, \qquad c : \mathcal{N} \longrightarrow \{T, F\}$$

4.4.1.1 Subscriber

A subscriber is initialized with a tag of interest and it is responsible for passing to the attached component all notifications that match the initialization tag. Formally, a subscriber $\mathcal{S}$ is defined as $\{P\}\mathcal{S}\{Q\}$ where P and Q are the pre- and postconditions of $\mathcal{S}$ respectively. $\mathcal{S}$ receives *sub* as initialization, where *sub* is a tag from Ω, i.e. $sub = (sub_a, sub_c)$; the pre- and postconditions are the following:

$$\begin{aligned} P &\equiv n \in \mathcal{N}, n \neq \epsilon \wedge sub_c(n) \\ Q &\equiv n' \in \mathcal{N}, n' = n \end{aligned}$$

Figure 4.5 (a) depicts a subscriber's component $\mathcal{S}$, where *sub* represents the tag of interest, n the incoming notifications and n' those notifications that match with *sub*.

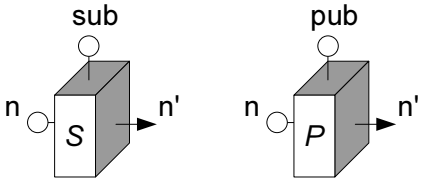

Figure 4.5: (a) Subscriber and (b) Publisher components

4.4.1.2 Publisher

A publisher is initialized with a tag of interest and it is responsible for publishing the output of the attached component into the underlying communication mechanism according to the initialization tag.

A publisher $\mathcal{P}$ is formalized as $\{P\}\mathcal{P}\{Q\}$ where P and Q are the pre- and post-conditions of $\mathcal{P}$ respectively.

$\mathcal{P}$ receives *pub* as initialization, where $pub \in \Omega$ and $pub = (pub_a, pub_c)$; the pre- and postconditions are as follows:

$$\begin{aligned} P &\equiv n \in \mathcal{N}, n \neq \epsilon \wedge pub_c(n) \\ Q &\equiv n' \in \mathcal{N}, pub_a(n') \wedge pub_c(n') \end{aligned}$$

In words, P states that the incoming notification must contain the same concept as the publisher was initialized to. Figure 4.5 (b) illustrates a publisher $\mathcal{P}$, where *pub* represents the initialization tag, n the output (in form of notification) to an attached component and n' those notifications that are delivered to the underlying communication mechanism.

4.4.2 Condition and Filter Service

As explained in Section 4.3.5, a filter can be seen as a specialization of a condition. Both take incoming notifications and verify a boolean predicate, selecting notifications by discarding those that do not evaluate to true.

In the next section, components are first formalized, and then a subscriber and a publisher are attached to them, forming a service that is ready to be combined with other components.

4.4.2.1 Filter

Filters are restricted to boolean predicates that involve only attributes contained within notifications.

Filter Component

A Filter Component $\mathcal{F}$ receives a triple $(sub, pred, pub)$ as parameter for initialization, but only uses *pred*. The arguments *sub* and *pub* are then used to initialize the communication components.

A filter component $\mathcal{F}$ is formally defined as $\{P\}\mathcal{F}\{Q\}$

$$\begin{aligned} P &\equiv n \in \mathcal{N}, n \neq \epsilon \wedge sub_c(n) \\ Q &\equiv [pred(n) \Longrightarrow n' \in \mathcal{N}, sub_c(n')] \vee \\ &\quad [\neg pred(n) \Longrightarrow n' \in \mathcal{N}, n' = \epsilon] \end{aligned}$$

Filter Service

A filter service $\mathcal{FS}$ is composed as in Figure 4.6, where a subscriber $\mathcal{S}$ and a publisher $\mathcal{P}$ are attached to the input and output interface of the filter component $\mathcal{F}$ respectively. As shown here, $\mathcal{F}$ is initialized with a triple $(sub, pred, pub)$ where *sub* and *pub* are respectively passed to the subscriber and publisher.

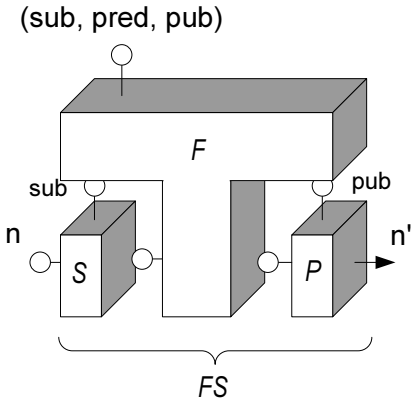

Figure 4.6: Filter service

Formally, the filter service is formulated as $\{P\}\underbrace{\mathcal{S}, \mathcal{F}, \mathcal{P}}_{\mathcal{FS}}\{Q\}$ where $\mathcal{S}$ is initialized with *sub*, $\mathcal{F}$ with *pred* and $\mathcal{P}$ with *pub*.

The predicate *pred* is defined as $\subseteq \mathcal{N}$, where pred : $\mathcal{N} \longrightarrow$ { T, F } (or { n $\in \mathcal{N}$ | pred(n) }) and the pre- and postconditions are these :

$$\begin{aligned} P &\equiv n \in \mathcal{N}, n \neq \epsilon, sub_c(n), sub_a(n) \\ Q &\equiv [pred(n) \Longrightarrow n' \in \mathcal{N}, sub_c(n') \wedge \neg sub_a(n') \wedge pub_a(n')] \vee \\ &\quad [\neg pred(n) \Longrightarrow n' \in \mathcal{N}, n' = \epsilon] \end{aligned}$$

The definition of this service can be interpreted as a piece of software that is responsible for discarding notifications that match *sub* and whose evaluation of the filter predicate does not evaluate to true. Those notifications that are not discarded must be republished according to the configuration of the publisher attached.

4.4.2.2 Condition

Conditions are boolean predicates that can involve (boolean) functions that are evaluated in an external system.

Condition Component

Assume that the world is modeled as a set $\Sigma = \{\sigma_1, \sigma_2, ...\}$ of states.

A condition component $\mathcal{C}$ is initialized with a triple $(sub, cond, pub)$ where $sub, pub \in \Omega$ and *cond* is a function from $\mathcal{N} \times \Sigma \rightarrow \{T, F\}$.

The component $\mathcal{C}$ has access to the current world state σ. The semantics of $\mathcal{C}$ are defined as follows:

$\{P\}\mathcal{C}\{Q\}$

$$\begin{aligned} P &\equiv n \in \mathcal{N}, n \neq \epsilon, sub_c(n) \\ Q &\equiv [cond(n, \sigma) \Longrightarrow n' \in \mathcal{N}, sub_c(n')] \vee \\ &\quad [\neg cond(n, \sigma) \Longrightarrow n' \in \mathcal{N}, n' = \epsilon] \end{aligned}$$

Condition Service

A condition service $\mathcal{CS}$ is composed as in Figure 4.7, and it is initiated with $(sub, cond, pub)$. As shown in the picture, this condition service is composed by attaching a subscriber and a publisher to the condition component.

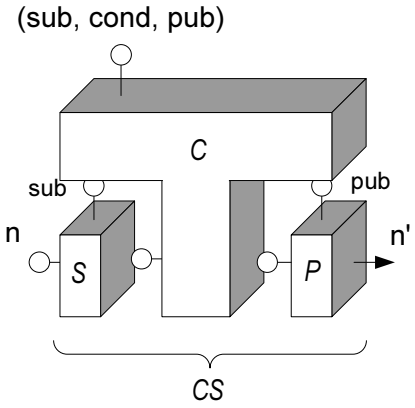

Figure 4.7: Condition service

The condition service $\mathcal{CS}$ is defined formally as $\{P\}\underbrace{\mathcal{S},\mathcal{C},\mathcal{P}}_{\mathcal{CS}}\{Q\}$ where $\mathcal{S}$ is initialized with *sub*, $\mathcal{C}$ with *cond* and $\mathcal{P}$ with *pub*.

$$\begin{aligned} P &\equiv n \in \mathcal{N}, n \neq \epsilon, sub_c(n) \wedge sub_a(n) \\ Q &\equiv [pred(n) \Longrightarrow n' \in \mathcal{N}, sub_c(n') \wedge \neg sub_a(n') \wedge pub_a(n')] \vee \\ &\quad [\neg pred(n) \Longrightarrow n' \in \mathcal{N}, n' = \epsilon] \end{aligned}$$

Similar to the filter service, a condition service is a piece of software that processes incoming notifications that match with *sub* and only those notifications are evaluated against the condition predicate. Notifications that evaluate to true are republished through the publisher attached to the condition component. The rest are simply discarded.

4.4.3 Action Service

The action service is responsible for binding attributes of incoming notifications to the defined action, transforming it into an executable form.

Action Component

Action component Λ is initialized with a tuple (sub, act) where $sub \in \Omega$ and act is a function that converts incoming notifications into executable actions.

$\Lambda = \{\lambda_1, \lambda_2, ...\}$ set of actions.

$\{P\}\Lambda\{Q\}$

$$\begin{aligned} P &\equiv n \in \mathcal{N}, sub_c(n) \\ Q &\equiv \alpha \in \Lambda, act(n) = \alpha \end{aligned}$$

Figure 4.8 depicts an action service $\Lambda\mathcal{S}$ that is initiated with (sub, act). This service is composed by attaching a subscriber to an action component.

$\{P\}\mathcal{S}, \Lambda\{Q\}$

$$\begin{aligned} P &\equiv n \in \mathcal{N}, sub_c(n) \wedge sub_a(n) \\ Q &\equiv \alpha \in \Lambda, act(n) = \alpha \end{aligned}$$

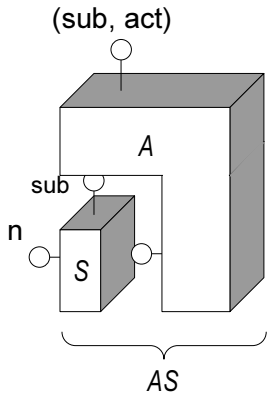

Figure 4.8: Action service

Notifications received at the action service should match with the *sub* used to initialize it. Those notifications are then bound to the action and transformed into an executable command that must be interpreted by the corresponding system.

4.5 Summary

This chapter described the architecture proposed in this work, which was complemented with a brief explanation of the role of event adapters. Afterwards, the elementary services involved were informally described and finally, the semantics of those services was formalized.

The proposed service-based architecture provides several advantages, namely: i) due to the uniformity of the (run-time) interface of all services they can be easily composed/-combined according to rule definitions in question, ii) the underlying publish/subscribe notification service (that uses concept-based addressing) provides flexibility to easily configure the flow of processing while maintaining subscription patterns at a higher-level, iii) because the terminology involved is self-documenting (concept definitions include a textual description) and since the method of communications is designed for interaction among heterogeneous services, components can be developed independently, iv) the evolution of the active functionality service is facilitated since the elementary services can be easily replaced or adapted, v) event adapters inject semantic events in the rule processing chain making possible the correct interpretation of their content.

Chapter 5

Event Composition

This chapter discusses and presents the problems associated with the complex event detection (or event composition) not only in centralized systems but paying special attention to distributed environments. Based on this analysis, the principles/design of an event composition platform that allows a flexible and clear definition of event operators is presented.

5.1 Characterization and Problem Description

Events determine to a large extent the expressive power of an active rule mechanism. Composition of events involves the occurrence of two or more primitive events and/or composite events (also called complex events). Usually, this composition relies on an event algebra [Dayal et al. 1988; Gehani et al. 1992; Gatziu and Dittrich 1993; Gatziu 1994; Collet and Coupaye 1996] that may include operators for sequence, disjunction, conjunction, negation, etc.

Most of the languages for defining complex events provide language constructors that are overloaded in the sense that the same situation can be described in different ways. This leads to a confusion of concepts that makes the understanding of an inherently complex area even more difficult. These confusions can be mainly attributed to a number of peculiarities and irregularities in existing event algebras [Zimmer and Unland 1999] and their implementations [Liebig et al. 1999].

Event composition in its general form depends on the ability to determine the order of occurrence of events. Logical clocks can not be used for this purpose because they can not represent timed real world events that are of common use in such kind of systems

(e.g. absolute and relative temporal events). Therefore, event order is achieved by using timestamps that are attached to event occurrences. The determination of the order of occurrence of events is important not only for event operators such as sequence, but also for all other operators since the consumption of events directly depends on it.

A solution to the problem of event consumption for the centralized case was presented in Snoop [Chakravarthy and Mishra 1994]. Here contexts specify the consumption mode of events, i.e. they are policy definitions that specify whether events should be consumed chronologically (*chronicle*), or whether the most recent instances should be used (*recent*). Besides these two obvious policies, two more have been defined.

In all cases mentioned above event compositions and event consumption modes implicitly assume a total order on events. These assumptions are quite reasonable for centralized systems for which they were defined but cannot be sustained in a distributed environment.

Inherent characteristics of distributed environments (like, the lack of a global time, independent failures, message delays, and simultaneity of happenings/events) increase the grade of difficulty of complex event detection. The characteristics imposed by this new environment invalidate the use of approaches designed for centralized systems. For instance, the reuse of the complex event detector unbundled from an implementation designed for centralized environments is not viable.

Typically event composition involves causal dependencies between real-word happenings or computations. Temporal order is a prerequisite for causal order. Therefore, potential causality can be detected (or excluded) when examining the order of event occurrences. However, occurrence time and global order of events in distributed environments can only be determined by an omniscient external observer. The notion of physical time is a well-known problem in distributed systems and there are global time approximations that can be applied to different distributed environments according to their characteristics. Generally speaking, in such kind of systems each site has a single physical clock which has its own local tick and is converted to local time by some software device. In order to compare the time of occurrence at remote sites, local clocks have to be synchronized (including the compensation for hardware clock skew and frequency drift). It must be noticed that in this kind of environment a total order of events cannot be guaranteed anymore and partial ordering must be dealt with.

For instance, Schwiderski [1996] adopted the 2g-precedence model to deal with distributed event ordering and composite event detection. This approach is suitable for closed networks where the granularity of global time-base g can be derived from a priori known and bounded precision of local clocks. Thus, it may not be suitable for the Internet where the accuracy and external synchronization of local clocks is best effort

and cannot be guaranteed because of large transmission delay variations and phases of disconnection.

The accuracy interval approach [Liebig et al. 1999] seems to be adequate for timestamping events in large-scale, loosely coupled distributed systems. Here local clocks are synchronized using the Network Time Protocol (NTP) [Mills 1990; Mills 1992] and timestamps are represented with accuracy intervals with reliable error bounds reflecting the inherent inaccuracy of time measurements.

Because of the partial ordered property of the global time approximation there may exist concurrent global event occurrences. Consider, for instance, the case of the recent consumption mode where the most recent event occurrence must be delivered when required. But now, the most recent may not be a single occurrence but a set of them and this fact must be treated accordingly.

Furthermore, event consumption in distributed environments must contemplate variable transmission delays, especially in the case of multiple, independent remote event producers.

The use of a *generic* event service requires that the semantics of event services that is presented to the application developer be not only formally specified but also unambiguous. If this is not achieved, it may cause applications to malfunction or behave nondeterministically.

5.2 Proposed Approach

In addition to defining an event algebra, systems that support composite events must also address the semantic issues associated with processing composite events. For example, the manner in which timestamps are generated and interpreted, and the way in which events are selected and consumed. Consequently, the adopted/implemented assumptions must be clearly exposed to developers and it must be possible for them to influence the service behavior by applying (predefined or user-defined) policies.

The active functionality service was designed to be used in a variety of scenarios which impact on the semantics of complex event detection. For this reason, the objective here is not to define yet another event algebra but to provide a flexible foundation/platform for event composition trying to unravel (as far as possible) the problems mentioned above by explicitly exposing to the developer the decisions/policies that must be taken under particular circumstances. On this basis, building complex event detectors should be a more predictable and easier task.

The proposal of such a complex event detection platform is motivated by the difficulties encountered in the implementation of different event operators where the semantics of the implemented operator (according to the event algebra) is difficult to verify/validate. The main reason is probably because of the number of problems that are implicitly addressed within the event operator implementation. For instance, the implementation of an event operator is normally implemented once for each different consumption mode. This makes adaptability to other environments very complex. Consider, for instance, the reuse of operators implemented for centralized environments where a total order of events was assumed and no transmission delays or failures were considered. Even though the logic of the event operator is the same, its implementation may be invalid demanding a re-design due to the new requirements imposed by the new environment.

With this in mind, simple but effective design principles are applied with the purpose of separating concerns to resolve the problems in isolation while providing a common framework in order to develop/implement event operators easily.

From the previous section three factors can be identified as crucial: the proper interpretation of time, the adoption of partial order of events and the consideration of transmission delays between producers and consumers of events. These factors are treated in the following subsections.

5.2.1 Proper Interpretation of Time - Timestamp Representation

As mentioned before the active functionality service can be used in a variety of scenarios and these scenarios may require different time assumptions and consequently different timestamp approaches. Moreover, each timestamp approach uses its own timestamp representation. But getting to a more abstract level the required functionality is basically the same in all cases: try to find out a correlation among timestamps. Considering all this, two main issues must be solved: i) how to represent timestamps in a flexible/open and "understandable"way, and ii) how to correlate them.

Because timestamps can be generated by different timestamp services, and additionally they are interpreted by every service/component involved in processing rules, a clear and unambiguous representation is required and for this purpose ontologies are qualified.

Consequently, timestamps and their related concepts must be defined in the ontology. Basically, an abstract timestamp concept must be defined and particular timestamp representations can be specialized for different scenarios and environments according to the time model (see Figure 5.1. Because the ontology is extensible, other time

stamp representations can be defined later leaving open the possibility to use the most appropriate timestamp mechanism in new scenarios.

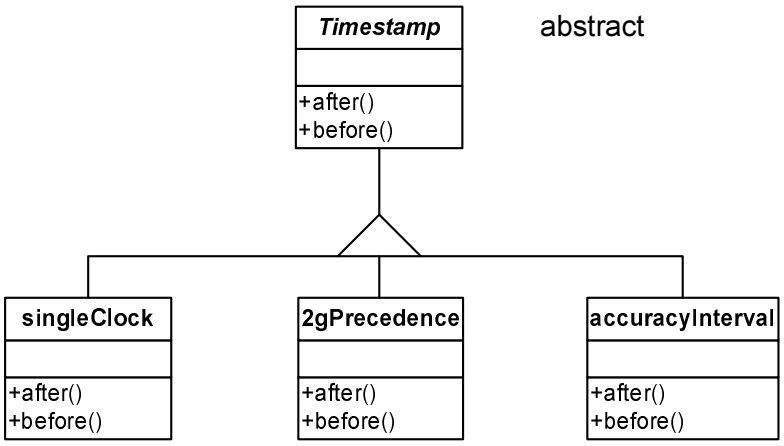

Figure 5.1: Timestamp representation approach

Following the concepts of data encapsulation the functionality of the abstract timestamp concept should include the correlation methods before and after while the data is maintained hidden. Notice that this data includes all assumptions made by the timestamp service that generates timestamp occurrences leaving no margins for misinterpretations. For instance, different implementations of the timestamp model can have differing assumptions. While one considers the amount of milliseconds since January 1st 1970, others assume January 1st 1980 as the base; or the case of time scale where some can assume Greenwich Mean Time (GMT) while others Universal Coordinated Time (UTC) that includes leap seconds[1].

The methods defined in the abstract timestamp concept (e.g. after) must be specialized for each particular model. Additionally, these methods throw exceptions when decisions cannot be taken transferring the decision control to another level. This is of particular interest for the handling partial order of events.

5.2.2 Partial Order of Events

In this work a partial order of events is assumed with the purpose of having a general case that can be specialized into cases where a total order of timestamps can be guaranteed (e.g. a single clock source).

[1]UTC (also known as civil time) is occasionally adjusted by one second increments to ensure that the difference between a uniform time scale defined by atomic clocks does not differ from the Earth's rotational time by more than 0.9 seconds [Time Service Department].

For instance, under the 2g-precedence model a total order can be determined only if timestamps are two or more clock ticks (g) apart. In the accuracy interval model the order of two timestamps is uncertain if they cannot be ordered (intervals overlap). As the order of timestamps cannot be decided in such cases, well defined actions should be taken.

With this in mind, correlation methods should include the possibility to throw an exception (e.g. cannotDecide) in order to announce such an uncertainty when comparing timestamps. In this way, application-specific decisions can be taken. That means that the underlying infrastructure is responsible for announcing this situation to a higher level of decision, keeping at application semantics level the resolution of such kind of circumstance.

With this, it can be guaranteed in all cases that: i) situations of uncertain timestamp order are detected and the action taken is exposed and well defined, and ii) events are not erroneously ordered.

5.2.3 Considering Transmission Delays, Failures, Order and Uncertainty

Usually, events coming from (event) producers are maintained in a temporary data structure before they are used for composition. Here, the EventList object encapsulates this data structure but additionally is responsible for subscribing to the event of interest. It must be noticed that EventLists are restricted to subscribe and maintain event instances of the same kind (those that are represented by the same ontology concept). For instance, an EventList subscribes only to StartOfAuction events. This restriction is only made with the purpose of facilitating the use of this kind of object as event operators simplifying the comprehension of event composition as it will be explained later in this chapter.

Since EventList is the intermediary between event producers and event composition, it seems to be an appropriate place to tackle the problem of transmission delays, failures at event producers, network failures, and also the order and uncertainty issues when working with event streams.

Some of the ideas presented in this section are related to the results presented in [Liebig et al. 1999].

This approach combines a window scheme with a heartbeat protocol to cope with node failures of producers and network failures like partitioning of the network or poor response time.

5.2.3.1 Heartbeat

Event producers are responsible for signaling events with a minimum frequency. If the event stream is less frequent or no more events occur at some producer node, the producer will generate an artificial heartbeat event with a two-fold purpose: i) increasing the sync-point reference (explained in the subsection below), and ii) announcing that the event producer is alive.

When a producer node crashes or the network is partitioned for long periods the event consumers could be blocked (possibly indefinitely). This problem is dealt with by using timeouts that in turn raise an exception. These kind of exceptions can be treated by a *failure handling policy*. This kind of policy can be specified by simply (re)using a pre-defined one or by treating the exception directly by the application in question by means of a callback method.

5.2.3.2 Window Mechanism

The window mechanism is used to separate the history of events (or event stream) into the *stable past* and the *unstable past and present* that still are subject to change. This separation is maintained by using a reference called here *sync-point*.

For composition purposes only events in the stable past are considered. That means, that the (time) reference used for selecting events from the event stream is the current sync-point. Notice that a time reference is required by consumption modes. For instance, the recent mode in centralized systems implicitly assumes the time reference as the current present. In the case of distributed environments the distinction between unstable and stable events is needed in order to consider transmission delays where some events that have already arrived should not be considered for composition until related events or sync-points from other producers arrive.

5.2.3.3 Consumption Modes

As mentioned before, consumption modes rely upon the temporal order of events. For instance, recent and chronicle select the latest (recent) or the oldest events (chronicle) out of the event stream. Therefore, events on the stable past are maintained (partially) ordered in the EventList according to the consumption mode criteria. EventList also implements the *event consumption interface* that is used for selecting and consuming events.

This interface defines the following methods: get(), returns the next event instance(s) in the event stream; getFromReference(ts) returns the next event instance(s) with respect to the reference passed as argument; consume() removes the next instance from the event stream; consume(ev) removes the event instance ev from the event stream; invalidateUpTo(ts) invalidates all events that are before the time reference ts; cleanUp() removes all event instances that are marked as invalid; and empty() returns true when the stable past has no event instances. Notice that the uncertainty policy is applied before these methods return a value.

The idea is that for each consumption mode these accessing methods should be implemented according to their selection criteria. For instance, in the case of a chronicle consumption mode the get method returns the oldest event in the stable past while in the case of recent it should deliver the latest one.

But this raises another issue mentioned before. Under certain circumstances related to the partial order of events, the get method could return not only a single event instance but a set of them. This issue can be handled by configuring the consumption mode with the required functionality. That is, if the event operator that gets/consumes events from this list requires single instances, then a *selection policy* can be applied. For example, a pre-defined policy can simply take randomly one event from the set and return it, or the whole set is passed to a user-defined policy that should decide what to do according to the application semantics.

As mentioned in Section 5.2.2, when correlating events the uncertainty problem could arise. To manage this situation, an exception is thrown and it should be treated appropriately. Again, the same principle of applying policies is applied. In this particular case an *uncertainty policy* should be specified. For instance, a policy can simply ignore the detected uncertainty and continue with the normal processing, or just discard one (or both) instances involved.

5.2.3.4 Putting it All Together

Figure 5.2 shows graphically the EventList where incoming events are maintained before composition. Events are separated into two categories to distinguish those events that can be used for composition. Notice that EventList can be configured with different policies (depicted with pins) in order to specify its behavior in cases of failures, multiple instances and uncertainty. Additionally, event instances are selected or consumed through the event consumption interface which provides a unique way to access event instances. In this way, these methods must be specialized in order to provide the order imposed by consumption modes. This approach simplifies the implementation of complex event operators, as it will be shown in the next section.

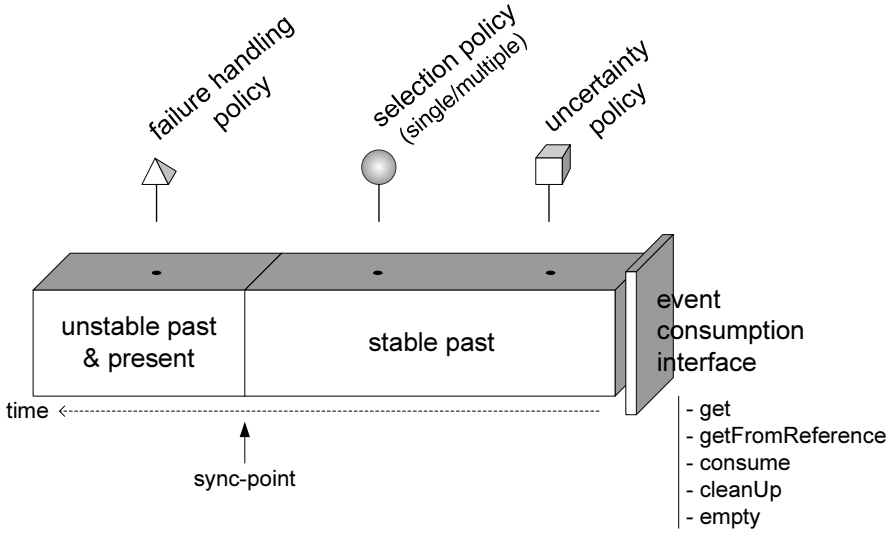

Figure 5.2: EventList: temporarily maintains event instances before composition

5.2.4 Event Composition

The basic infrastructure of the complex event detector service is based on the principles of components and containers. Components are the event operators that are plugged into containers (also called compositors here). The container itself is the complex event detector kernel which controls the event detection process. As shown in Figure 5.3, the container has attached, in this case, two EventLists that play the role of event operands. These, in turn, are responsible for subscribing and maintaining events. Additionally, they are configured with appropriate policies according to the definition of the complex event that must be detected.

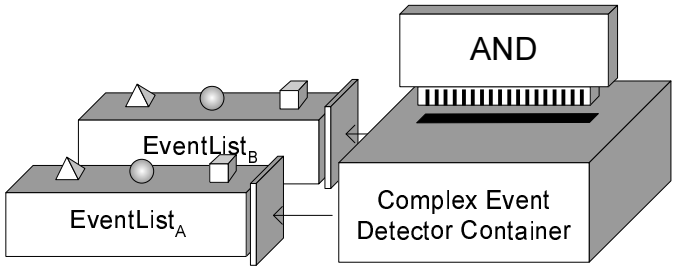

Figure 5.3: Abstract view of an event compositor

As mentioned above, the compositor controls the detection of complex events. A pseudo-code of its behavior is presented in Algorithm 5.1. As every server, it implements an infinite loop. Once in the loop, it waits for a sync-point that are announced

by the attached operands. Sync-points are maintained in eventsForProcessing queue. If eventsForProcessing is empty the container blocks and it is woken up when incoming events or heartbeats arrive moving, as a collateral effect, the sync-point forward. This is the moment where possibly new events form part of the stable past and therewith open the possibility to detect new complex events. Therefore, the container calls the plugged component – the event operator logic – in order to evaluate if the complex situation in question can be detected. If so, the composite event is returned, additional (contextual) information can be added, and finally the detection is announced through the notification service with the purpose of notifying other interested parties (e.g. other compositors or services). Afterwards, it cleans up involved operands to discard those event instances that cannot take part anymore in other compositions. Notice that in this work event instances are disseminated by using a publish/subcribe approach intrinsically replicating the event instance in question at every consumer. As a consequence, at every compositor can be simply analyzed if event instances can be discarded from the list without any other consideration.

Algorithm 5.1 Compositor behavior

```
while ( TRUE ) do
  eventsForProcessing.get(); // block if empty
  try
      compEv ← evOperator.evaluate();
      // compEv.addContext( <consumptionMode,...> );
      // compEv.opData.add( <detectionTime, ...> );
      notifSrv.publish( compEv );
      foreach operand op do
         op.cleanUp();
      end for
  catch NotYet()
      // nothing to do, just wait for incoming events
end while
```

A sequence diagram is presented in Figure 5.4 showing the interaction among the entities that participate in the composition and how the compositor orchestrates the whole process. For the sake of simplicity, the communication of the event operator logic with the operands (EventLists) are not depicted. Notice that the interaction between operands and the compositor is asynchronous. This allows the container implementation sufficient freedom to decide when is the proper moment to perform the complex event detection.

Up to this point the behavior of the compositor was presented showing that it is in charge of deciding when to evaluate and what to do after detection. Now, it is the time

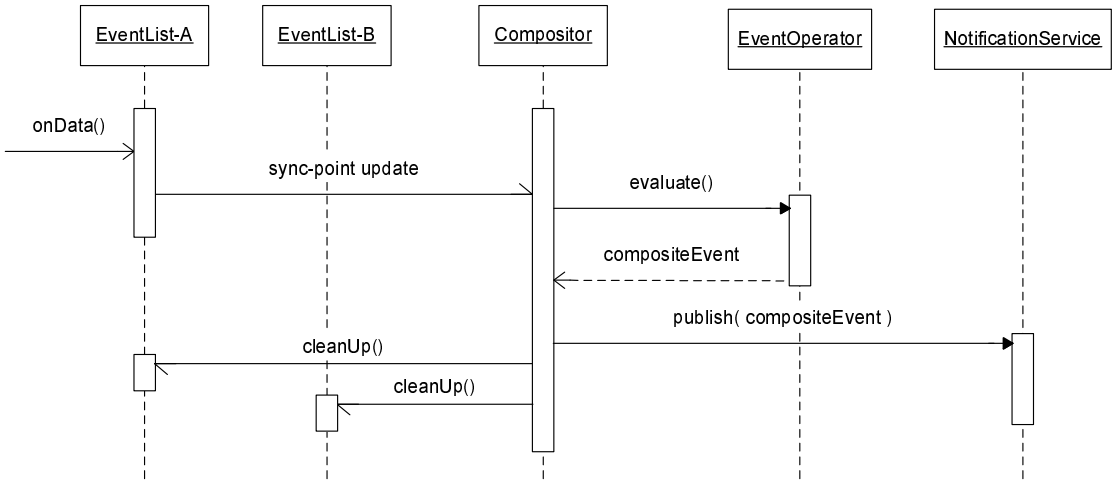

Figure 5.4: Sequence diagram of the interaction among entities participating in an event composition

to present the component that is plugged into a container. This component is responsible for the logic of the event operator, i.e. how to detect the situation expressed by the event operator. For this purpose, components implement the method evaluate that provides containers a point of contact that is called at the proper moment. Components specialize the EventOperator class by re-writing the evaluate method according to the operator they represent. In this way, other information, like the references to the operands, can be passed to operators without much effort. A class hierarchy depicting this situation is presented in Figure 5.5.

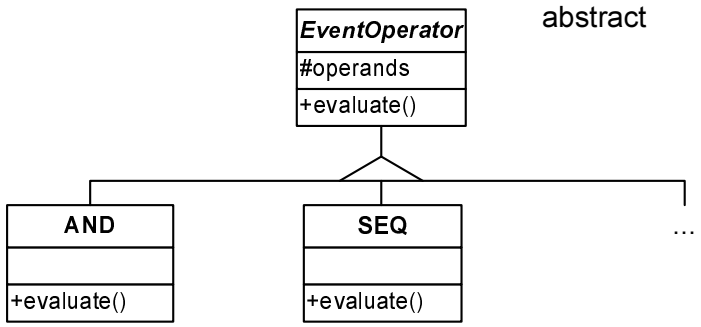

Figure 5.5: Operator's class hierarchy

The pseudo-code of the evaluate method that in fact represents the logic of the AND and SEQ operators is presented in Algorithms 5.2 and 5.3 respectively.

Notice that the logic of operators concentrates on detecting the situation of interest

Algorithm 5.2 evaluate method - Logic of the AND event operator

```
if ( (not operandA.empty() ) ∧ (not operandB.empty() ) then
  a ← operandA.consume();
  b ← operandB.consume();
  return new AND( a, b );
else
  // exception
  throw NotYet();
end if
```

isolating the operator's logic from other aspects which are now the responsibility of other entities. In particular, the order in which event instances are selected (consumption mode) is resolved by the EventList which implements the proper access criteria through the event consumption interface. Moreover, the consideration of failures and transmission delays, as well as uncertainty issues are solved at the EventList by applying pre-configured policies.

Algorithm 5.3 evaluate method - Logic of the SEQUENCE event operator

```
if ( (not operandA.empty() ) ∧ (not operandB.empty() )) then
  a ← opearandA.get();
  b ← operandB.getFromReference( a.ts );
  if ( b ≠ nil ) then
    operandA.consume( a );
    operandB.consume( b );
    return new SEQ( a, b );
  end if
end if
// exception
throw NotYet();
```

If the complex event of interest cannot be detected, the evaluate method is responsible for throwing the NotYet exception with the purpose to make this situation explicit. In fact, this exception is caught by the compositor which acts accordingly.

Under these circumstances, the logic of event operators can be used with different consumption modes without requiring to have an implementation of each operator for each consumption mode (as is the case of many operators in current prototype implementations [Schwiderski 1996; Ma and Bacon 1998; Zimmermann and Buchmann 1999; Chakravarthy et al. 1999; Dittrich et al. 2000]).

5.2.4.1 Composing Composite Events

Because of its uniform design, compositors can cooperate in the detection of other composite events. Particularly, compositors publish detected events in the same way primitive events are published. Thus, the output of a compositor (a composite event) can be used for subscription of other parties. Consequently, this can be seen as an abstract tree where primitive events are injected at the leaves and compositors are located in the internal nodes and at the root. Detected events are pushed to the upper layer in the tree by using the publish mechanism. The whole complex event is detected once an event is published at the root of the tree.

Consider for instance the complex event ((A AND B) SEQ C) that is graphically presented in Figure 5.6. In this particular case, the $\mathsf{EventList}_A$ subscribes for A events, while a similar situation occurs for B and C events. Notice that the compositor that evaluates the AND operator has attached an EventList for A and another for B. This complex event corresponds to the subtree (A AND B). The compositor responsible for the detection of the SEQ operator has attached the $\mathsf{EventList}_C$ and the $\mathsf{EventList}_{AandB}$ which subscribes to event occurrences of the kind (A AND B).

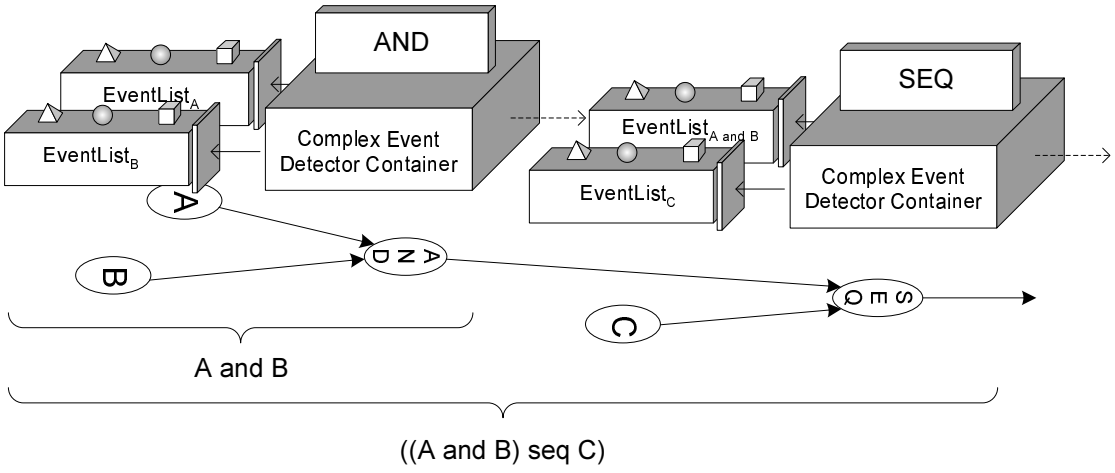

Figure 5.6: Composition of complex events

Incoming A and B event instances cause the evaluation of the AND operator and when composite events are detected they are immediately published. The arrival of event instances at $\mathsf{EventList}_{AandB}$ and $\mathsf{EventList}_C$ may cause the evaluation of the SEQ operator. As with the AND operator, if the complex event is detected, it is published. In this particular case, the root of the tree is reached meaning that the whole composition of complex event was detected.

Looking at a higher level of abstraction and because compositors use the publish subscribe mechanism, many other compositors can subscribe to a composite event. In this way, the tree structure is now an oriented graph.

5.3 Summary and Conclusions

As mentioned at the beginning of this chapter, event algebras and their implementations have peculiarities and irregularities. In these cases, the implementation of operators not only implements the operator semantics but also the consumption mode in question. This made, for instance, the unbundling approach infeasible in particular when trying to reuse the event compositor in distributed environment for which it was not designed for.

In contrast to the approaches mentioned above and in order to simplify or at least to understand the problems involved, a separation of concerns was conducted obtaining as a result: proper interpretation of time, consumption issues, handling of failures, consideration of transmission delays, and the proper representation of timestamps. For instance, ontologies are used to represent different timestamp models allowing their proper interpretation; a combination of a window mechanism and a heartbeat protocol is used to cope with transmission delays, network and producer failures; and the use of policies to configure the actions that must be taken in case of anomalies.

A flexible platform for event composition was presented relying on the principles of containers and components. Here event operators play the role of components that are in turn plugged into the container which controls the event detection process. In this way, hard-wired operators are avoided by easily implementing the operator's functionality (or logic) as a component without considering other problems (e.g. transmission delays, failures) that in fact are solved properly elsewhere.

It must be noticed that in the worst-case the frequency of heartbeats is the one that dictates the detection pace. This is because only instances in the stable past can be considered for composition.

There are still open issues to be investigated like the verification that other consumption modes such as, continuous, or cumulative can be also supported within this platform; a detailed analysis of the impact of the heartbeat frequency on the event composition detection; a testbed to simulate real situations by generating events at event producers and by inducing failures and other difficulties in order to study how compositors and policies behave. Additionally, a formal validation of this approach is desirable.

Chapter 6

Prototype Implementation

This chapter includes details about the implementation of a prototype that demonstrates most of the issues discussed and proposed in this dissertation. The chapter begins with a description of the implementation of ontology concepts followed by a summary of characteristics of the service framework that is used as platform for the active functionality service prototype. The chapter is completed with a description of the implementation of elementary services, the concept-based addressing approach, adapters, plug-ins, how rules are defined and how the ECA-Manager works.

6.1 Ontology Representation

Ontology concepts are specified using the Java programming language as described in [Bornhövd 2000]. This ontology representation has been successfully used in the MIBIA project [Bornhövd and Buchmann 2000].

In order to simplify the creation of concept instances, an Ontology API is defined providing a common interface to interact with the ontology. This is particularly useful for the development of adapters.

6.1.1 Specifying Ontology Concepts with Java

Ontology concepts and their relationships are implemented using the Java language avoiding any impedance mismatch between programming language and ontology specification language, and allowing the shipping of ontology concepts between different platforms without any further transformations. In addition to data portability, Java

supports code portability which is important because conversion function behavior can also be portable.

Concepts are defined as classes and each class contains an informal textual description, and a formal computer interpretable specification of the concept and its semantic relationships with other concepts (e.g. generalization/specialization, aggregation). Additionally, concept specifications may contain concept-specific functions like comparison operators or conversion functions.

Figure 6.1 presents a graphical representation of the implementation of semantic concepts. Simple semantic objects consist of: a slot to keep the actual data value, and a list of semantic objects representing its semantic context. Since aspects specified in the context are not determined by the concept class, the context information may vary between instances of the same concept.

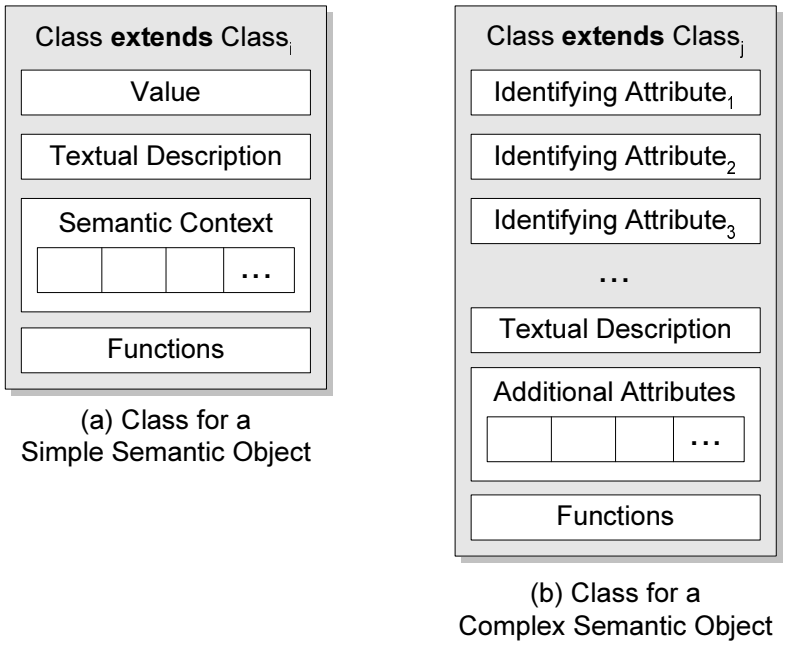

Figure 6.1: Representation of Ontology Concepts with Java

For concept classes used by complex semantic objects, the specification consists of: slots for identifying attributes, and a list of non-identifying attributes. Attributes can be understood as forming part of an is-part-of relationship. The content of the list of non-identifying attributes may vary between different instances of the same concept.

Concept classes are organized in packages that contain sets of concepts belonging closely together, such as metric system, basic representation, etc. Packages support a modular

organization of ontologies as packages and sub-packages which in the implementation correspond one-to-one to Java packages. Concepts (classes) from other packages can be referred to with the Java import statement.

Figure 6.2 shows the organization of ontology concepts in three levels as defined in Section 3.1.1. Each level, that can be seen as a package, contains different sub-packages. The *Basic Representation* package contains concepts like String, Number, Date, etc. organized into the Represent, EngMath and Calendar sub-packages. The *Infrastructure-specific* package covers those concepts closely related to the active functionality service and it is organized in the following sub-packages: ECARule that contains all concepts related to ECA-rules like Condition, Action, Event, TemporalEvent, etc.; Notification comprises concepts associated to messaging, like, Notification, OperationalData, etc.; Service consists of terminology concerned with the service description like ServiceURL, Binding, etc; and Timestamp includes the representation of different timestamp models and their corresponding concepts like Timestamp, ClockSource, etc. *Domain-specific* concepts can be plugged in depending on the problem domain. For instance, the Auction package contains concepts like BidAmount, AuctionDeadline, AuctionItem, etc.

6.1.2 Ontology API

Ontology concepts can be instantiated simply by calling their constructor or by using the Ontology Application Programming Interface (API). This API provides application developers with methods that facilitate the creation of instances of ontology concepts. Additionally, it offers the possibility to pre-instantiate semantic objects with default contexts and values. In this way, for instance, an application adapter can set the contextual information of the data that is going to be "exported". Probably this contextual information is static with respect to a particular application and it is reasonable to set this context once and attach it to every concept instance of this kind. For example, consider the case where an application assumes all prices in U.S. Dollars or the format of the date as "MM/DD/YYYY". In these cases, when this kind of data is exported application assumptions must be added. Instead of adding it each time the information is "exported", it can be defined once and is then automatically attached when the data goes outside the application bounds. For this purpose, the API provides the possibility to setup default contexts and values for all concepts used by an application. This API is the basic building block for developing ontology-based software.

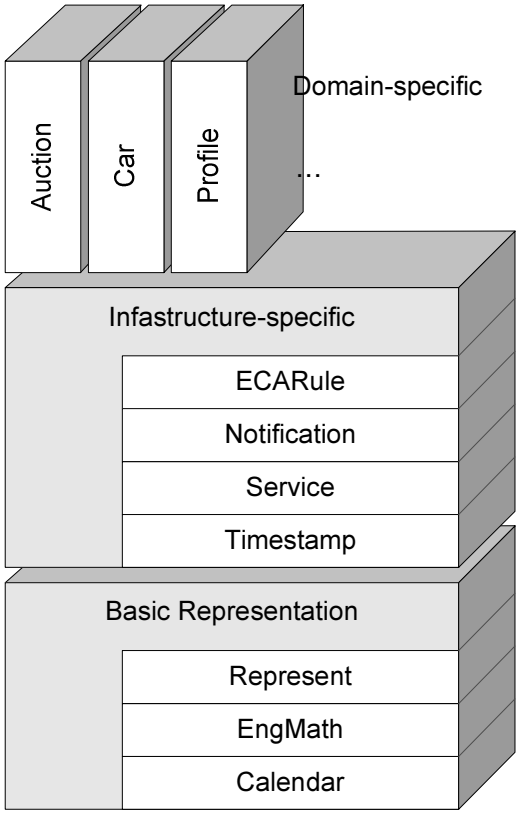

Figure 6.2: Organization of ontology concepts as packages

6.2 Service Platform

Essential to any service platform is the technology that allows developers to create streamlined, compatible services. Toward this end a platform should provide the following:

- Well-defined contracts that ensure services are accessed and managed in a unified and consistent manner.
- Service-dependency management that defines exactly how services interact with each other and the framework.
- Management software that allows developers to dynamically configure, maintain, and deploy services.

In addition, a service platform should have an inherently open architecture allowing its functionality to be leveraged by a variety of applications (e.g. databases, application

servers, and embeddable hardware programs).

At the time the implementation of the prototype was started, the Core Service Framework (CSF) [HP Bluestone 2001] was one of the few service platforms that were developed and in beta phase of testing. This platform was the appropriate starting point for a prototype implementation fulfilling most requirements of the prototype. Its description is presented in the following subsection.

6.2.1 The Core Service Framework

The Core Service Framework [HP Bluestone 2001] is a component framework designed to facilitate the development of services from a set of independent but cooperating services. The framework defines the life cycle of these services as well as how they may be located and interact with each other. In addition, the framework supports the run-time configuration and management of these services.

The fundamental principle of this service architecture is the separation, or layering, of logic. There are two types of logic that must be developed for a service: service logic and functional logic. Service logic is the code that represents the operation of the service. Functional logic is a wrapper that makes the service logic work within the framework. In a properly designed service, the service logic should represent 90 % of the development effort so that the resulting code can be reused with any framework.

6.2.1.1 Organization

The CSF is organized based on the concepts of Embeddors, Kernel, and Partitions (see figure 6.3). *Embeddors* are the "shell" that creates a service environment within a device. An embeddor is used to communicate with the external environment, providing the information it obtains to the services that it hosts. The primary role of an embeddor is to create, maintain, and eventually destroy the instance of the kernel that resides within it.

The *Kernel* exists within an embeddor and provides fundamental functionality to the other framework components. Several interfaces are provided by the kernel to other framework components in order to expose only those capabilities required by each component type. In most ways, the kernel can be considered as partition and as service within the framework. It provides arbitration for all the framework's resources.

A *Partition* is a specialized service that provides a way to enclose a set of services. It provides very little functionality other than manipulating the life cycle of the service it

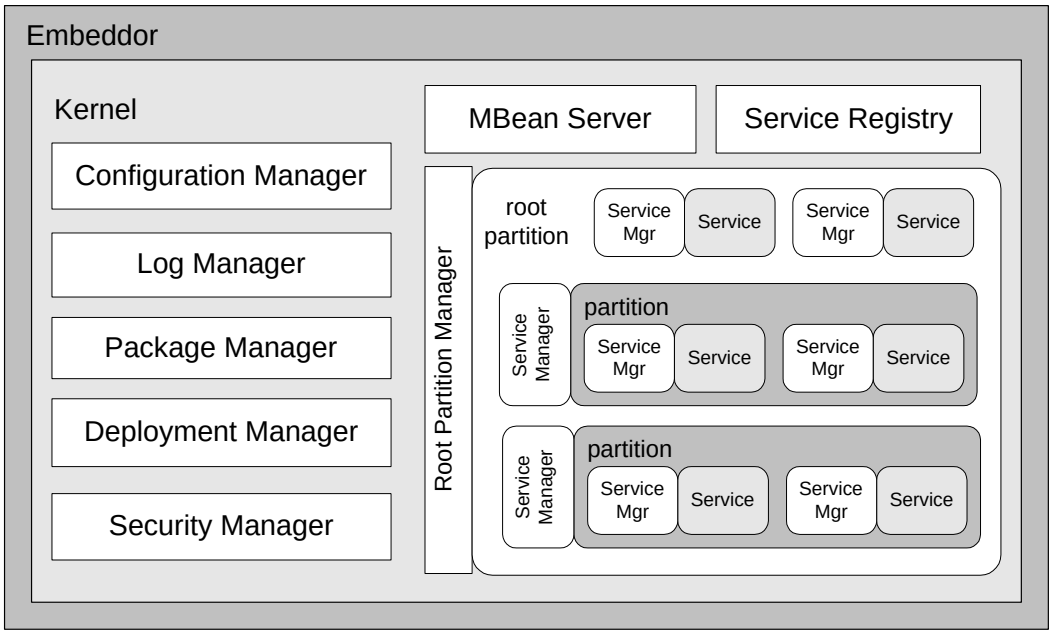

Figure 6.3: CSF architecture

contains. The principal role of a partition is to control the visibility of services within the framework. For instance, they are useful to support multiple versions of the same service within a single framework, or to define different security roles for each partition.

As depicted in figure 6.3, the *Service Manager* plays an important role, being responsible for creating, managing and destroying services. Each time a service is deployed, the kernel creates a separate service manager for that service, which is basically in charge of: a) creating a single service, b) creating a class loader for that service, and c) registering the service in the service registry.

Services provide an encapsulation of well-defined functionality. They locate one another through the service registry provided by the kernel. Services are deployed in a partition and run within it. Services can implement any of the interfaces that have been developed for the framework. These interfaces are used to tell the framework which contracts the service is willing to participate in.

As mentioned before, each service in the CSF is registered in the *Service Registry*. Client services can make use of other services by looking up those services in the registry. Clients can access the service functions that are exposed through the service's interface.

An MBean is a service-generated object that exposes run-time management and configuration behavior to the framework. A service can create as many MBeans as it needs to

obtain the management and configuration capabilities that it requires. These MBeans are registered with the *MBean Server*. External distributed management functionality is facilitated by MBeans.

The CSF handles configuration, logging, packages, security and deployment management to free developers from this overhead. This is done through its managers.

6.2.1.2 Service Life Cycle

The interfaces and contracts defined by the framework provide a clear delineation between the functionality provided by the framework. The idea is to allow a service developer to select and implement only the interfaces and contracts required by the functionality of a particular service.

At runtime, the service manager invokes callback methods (interface implementation) on the service instance when appropriate state change events occur. The most important states of a service life cycle and their activities are described in Table 6.1.

State	Activities
Resolution	Initialization of Service Manager; class loader is created; service Context is created.
Initialization	Service implementation is instantiated; service is registered with the Service Registry; the service is provided with the Service Context; the service is asked to initialize itself.
Start	Logical references to dependent services are resolved; the service is asked to start itself.
Reconfiguration	The appropriate service method is invoked; the service uses the configuration manager through the service context to retrieve a configuration object.
Stop	The service is asked to stop itself; all dependent services acquired by the service are released.
Destruction	If the service is running, it is stopped; all dependent services acquired by the service are released.

Table 6.1: States of services' life cycle

6.2.1.3 Service Development

From the service development perspective, the *Service Context* is responsible for interfacing with the framework components on behalf of the service. It provides a service with access to framework resources (e.g. the service registry). The service manager, the service itself, and its context work in tandem to manage a service and provide it with access to framework resources.

Services within the framework typically depend on other services to provide useful functionality but might not need to specify which service provides it. The platform expects services that can be used by other services to expose an interface. This interface is the critical piece of information that defines the functionality a producer service provides to a client service.

CSF provides a method on the service context that allows a service to be found by specifying the interface required along with other optional criteria. Additionally, services can be searched by name, by version number, and by extra attributes.

6.2.1.4 Service Deployment

Deployment is the process of installing a service in the platform. This process involves three tasks:

- Defining the service's static properties in a service descriptor file.
- Specifying the service's dynamic properties. These are properties that can change when the service is installed. In this way, they are initialized with the corresponding values.
- Invoking the installService method so that the provided service properties can be used to install the service.

This process involves interactions between most of the architectural components of the platform.

6.3 ECA Elementary Services

The active service prototype is built on top of CSF because of the facilities offered to develop and deploy services. The prototype implementation basically follows the

main ideas of the service formalization presented in Section 4.4. As shown in the formalization, three main components form part of elementary services: the service logic itself, the publisher and the subscriber.

In this section the implementation of the underlying publish/subscribe mechanism is introduced first since this mechanism is the communication mechanism among services. After that, it is shown how elementary services are developed beginning with the organization of classes and followed by the implementation of concrete elementary services.

It should be noticed that all method arguments used in this chapter form part of the infrastructure-specific ontology[1].

6.3.1 Notification Service

There are several aspects that must be considered when implementing a notification service, such as the cardinality of producers and consumers (1:1, 1:n, n:m), synchronous or asynchronous communication (method invocation, queues, store and forward), guarantee of message ordering (FIFO, no order guaranteed), message encryption, reliability of the service (message is received at least once, only-once, acknowledgments, no guarantee), time-to-live of messages, maintenance of events in an event log, delivery mode (scheduled, immediate, or delayed), etc.

The characteristics of the event dissemination mechanism required in this work include:

- asynchronous communication,
- publish/subscribe support,
- message delivery in local and wide area networks like the Internet,
- support of a broad set of platforms (including mobile environments) and
- transaction support

For the underlying communication mechanism TIB/Rendezvous [TIBCO] is used which matches most of the requirements listed above providing an efficient multicast dissemination of messages and a publish/subscribe mechanism based on subject-based addressing. On top of it the notification service was built that is enhanced with concept-based

[1] A list of infrastructure-specific concepts can be found in Appendix B - Section B.2.

addressing. The integration of transactions and notification services, necessary to support coupling modes, is being developed in our group under the X^2TS project [Liebig et al. 2000a].

The publish/subscribe paradigm naturally decouples consumers and producers providing the possibility to reach a set of interested consumers by sending a message only once. Additionally it provides location transparency which in this context allows services to be upgraded, moved or replaced without having to modify any programming code.

6.3.1.1 Concept-based Addressing

As mentioned in previous chapters this work introduces *concept-based addressing*. To put this in correspondence with its name, subscriptions are made based on the concepts defined in the underlying ontology. In this way, consumers do not need to take care of proprietary representations and all participants use a common vocabulary not only for its physical and structural representation but also for making explicit its assumed meaning.

Before getting into details about concept-based addressing, subject-based addressing as used by TIBCO must be introduced. Subjects define a uniform name space for messages and their destinations. A subject is associated to each message. Subject names consist of one or more elements (usually a string) organized in a tree by means of a dot notation. Subjects are used to direct messages to their destinations, so applications can communicate without knowing the details of network addresses or connections and the location of message consumers becomes entirely transparent without requiring a name service. Moreover, new message producers and consumers can be introduced at any time.

In this prototype, the subject name space is organized in two main parts. The first one, is to provide control of the destination of notifications (if needed). This control part is used to concatenate services in the service chain. The second part is used to capture the notification content (in particular the event in question) to allow more powerful subscription.

Coming back to the implementation of the concept-based approach, concept instances are mapped to the subject name space, where the concept name forms part of the subject. It is also possible that attribute values of a concept constitute part of a subject in order to allow a more specific subscription. In particular, identifying attributes of concepts are candidates to form part of the name space. Figure 6.4 shows how event content is virtually mapped into the subject tree and how a particular subject instance

is derived. This way, the destination of notifications is determined by self-contained information.

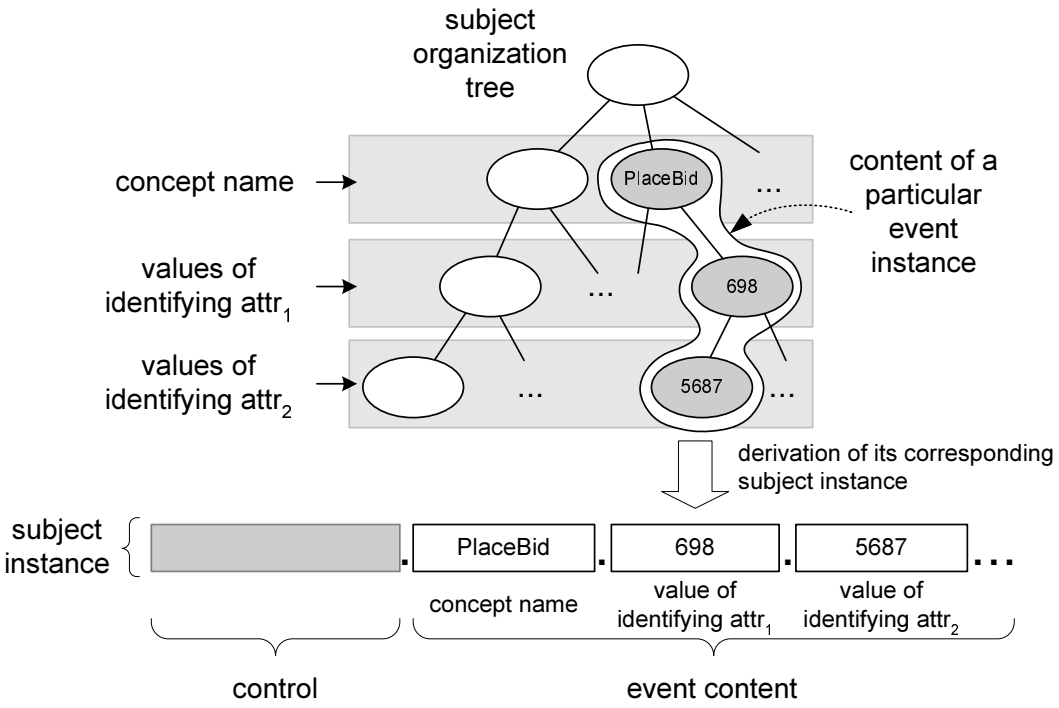

Figure 6.4: Subject organization and subject instance derivation

Derived subject instances are then attached to their corresponding messages and used by the underlying communication mechanism to reach message consumers. These consumers express their interests by specifying concepts of interest where wildcards can be used for that purpose.

Both parts of the subject organization are configurable. That is, the depth of the subject organization tree (i.e. how many identifying attributes are included in the subject), as well as the number of fields that form part of the control subject can be configured. All this information and the name space organization is maintained in a repository.

6.3.1.2 Publisher & Subscriber

Two main components of the notification service are distinguished from the client perspective: the *Subscriber*, who is responsible for the client subscriptions (messages of interest), and the *Publisher*, who permits the publication of messages. These components expose two main kind of interfaces as depicted in Figure 6.5.

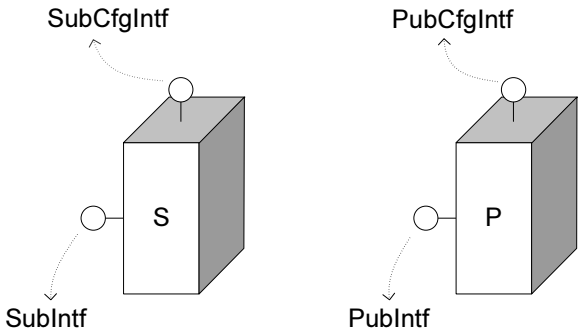

Figure 6.5: Graphical representation of a subscriber (S) and a publisher (P)

Subscribers provide clients with the possibility to express their interests through the SubCfgIntf interface. In this way, subscribers can make explicit their interests at the ontology level, where subscriptions are made by passing pre-instantiated concepts that can contain specific values and/or wild-cards. Clients receive incoming notifications by means of the service run-time interface SubIntf. This interface is used by the underlying notification service in order to deliver notifications to the consumer.

Publishers make visible the PubCfgIntf interface to provide clients with the possibility to pre-configure part of the subject, in particular the control part. This is useful to "route" messages to a specified destination. Its use is optional. Through the PubIntf interface publishers allow clients to effectively publish events in the semantic pipeline. Its publication involves some steps that are graphically presented in Figure 6.6. A publisher is responsible for deriving the subject from the event content (1a) which must be joined with the control subject if any is defined (1b). The notification message is created (2a) and the corresponding operational data (like source, sending timestamp, etc.) is added (2b). After that, the publisher is responsible for attaching the resulting subject to the message and finally passing it to the underlying communication mechanism (3).

Clients of a notification service can be organized into two classes: i) those clients that are exclusively interested in sending messages (pure publishers), and ii) those clients that are interested in receiving and publishing messages.
Putting this in context, there are services like event adapters or alarm services that attach only a publisher component, and others that attach both, for instance, the condition and filter services. Details about the implementation of these services are described in the following section.

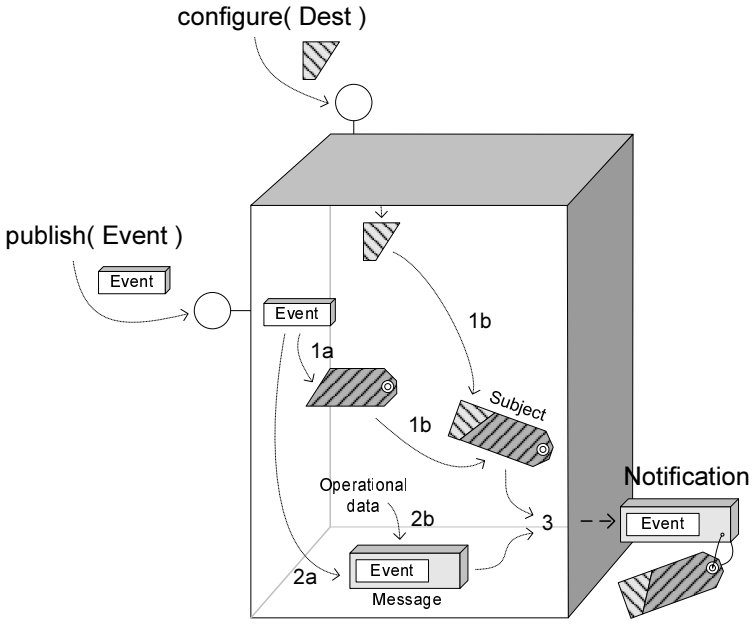

Figure 6.6: Publisher: Steps involved in publishing an event

6.3.2 Class Organization of ECA Elementary Services

As mentioned in the previous section, the service logic of a properly designed service should represent 90 % of the development effort. This code must be able to be reused within any service framework. To this end the class model of the service implementation was designed to encapsulate the necessary functional logic (the complementary 10 %) that depends on the CSF platform. As depicted in Figure 6.7 classes are organized in three layers.

On the top of the figure, the abstract class BasicSrv implements the interfaces (or contracts) directly associated with the CSF platform. In the second layer, the abstract class ECAElemSrv extends the basic service class by implementing the interfaces related to the ECA architecture. Finally, at the bottom, the third layer contains concrete elementary services (for instance, ConditionSrv, ActionSrv) that are implemented by extending the abstract class ECAElemSrv.

Taking a closer look, the BasicSrv class implements CSF's service, initializable, startable, reconfigurable, stoppable, and destroyable interfaces. The methods defined in these interfaces are the callbacks that the framework uses to notify life-cycle state changes of the service to the service implementation itself. The implementation of the Service interface is mandatory and its setServiceContext method is the first method invoked by the CSF framework passing the service context as an argument. This context allows

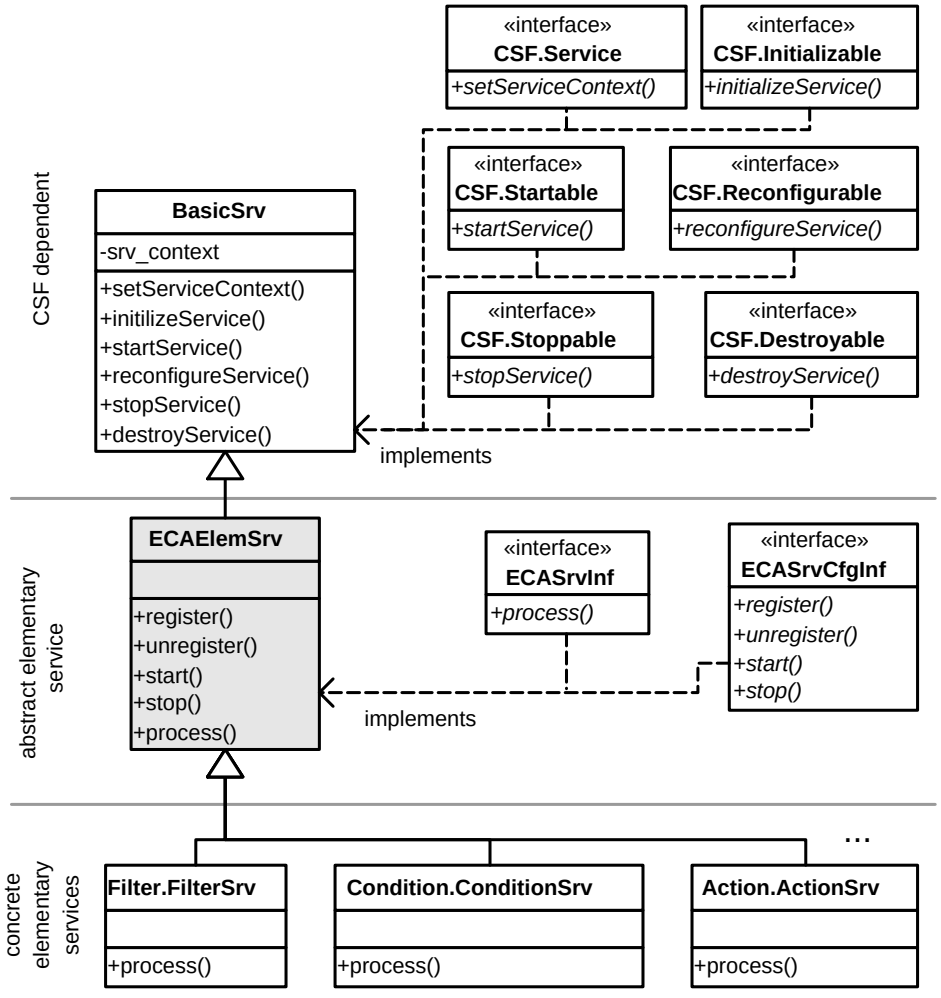

Figure 6.7: Class organization of the ECA elementary services

the service implementation to interact with the service framework, and for this reason it is assigned to the instance variable srv_context where it is maintained in order to facilitate access to the framework at any time.

As mentioned before, the abstract class ECAElemSrv extends the basic service class and implements the interfaces related to the ECA architecture, namely the configuration interface ECASrvCfgIntf and the (run-time) service interface ECASrvIntf. Both interfaces are very simple and they are the foundation of the straightforward composition of elementary services. A schematic view of this class is shown in Figure 6.8 where both interfaces are depicted by means of lollipops. The socket tries to graphically represent that this class implements elementary ECA infrastructure (functional logic) and that the service logic must be plugged-in in order to make it work. In the rest of this section and depending on the description of other services this picture is going to be completed accordingly.

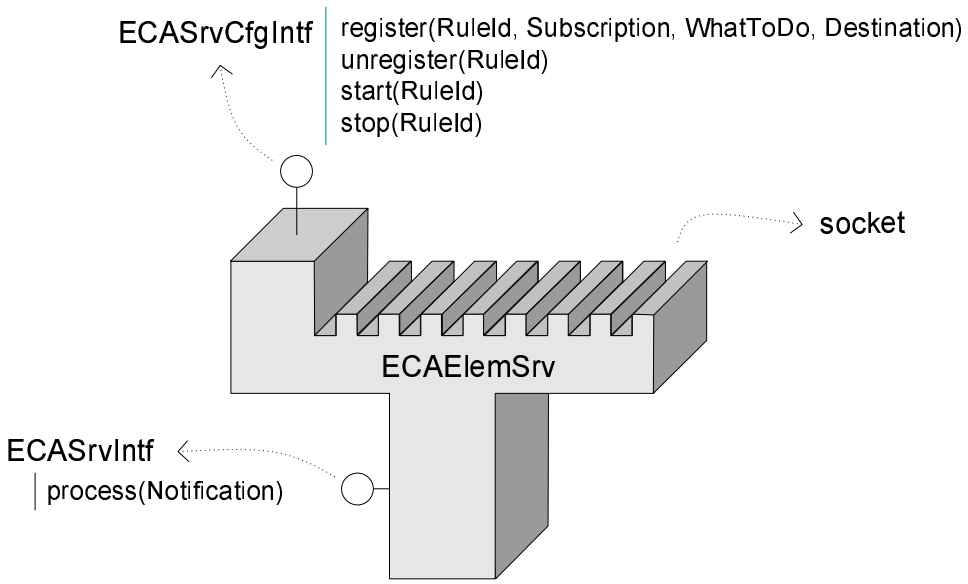

Figure 6.8: Schematic view of the abstract class ECAElemSrv

The ECASrvCfgIntf interface, as its name indicates, defines four methods to configure the service: register, unregister, start, and stop. The register method has four arguments: RuleId identifies the rule in question; Subscription indicates what to subscribe to i.e. the notifications of interest; WhatToDo represents a "generic object"–in this case an expression (for instance, a boolean predicate) with references to notification attributes; and finally the last argument, Destination contains information related to the output of the service, such as where processed notifications must be published if needed. The other three methods of this interface have only RuleId as argument. The unregister

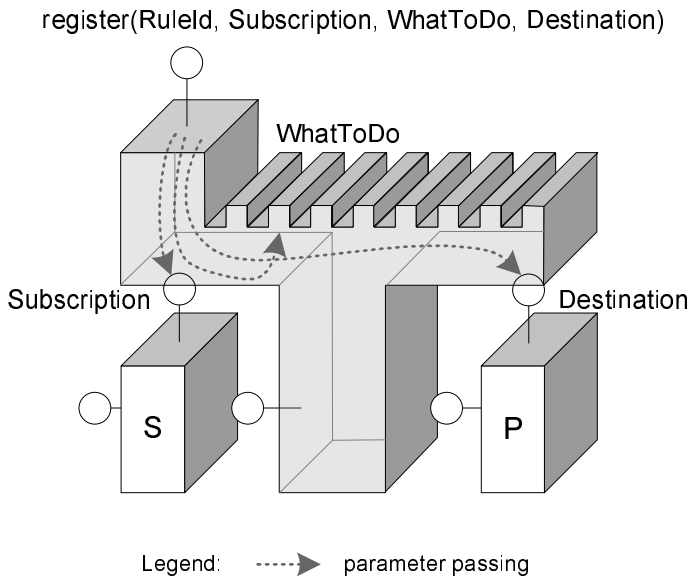

Figure 6.9: ECASrvCfgIntf's register method in context

method simply unsubscribes its interests, and it clears other related information, such as the publisher configuration. The start and stop methods are used to start and stop processing notifications respectively.
Figure 6.9 presents the register method in context. As it can be seen in the picture, a Subscriber and a Publisher are already attached. The register method configures the subscriber with the corresponding notification interests, and it pre-configures the publisher with the purpose of simplifying the publishing task. This configuration is achieved by passing through the configuration interface the Subscription and the Destination arguments to the subscriber and publisher respectively. The WhatToDo argument is assigned to an instance variable that can be accessed by concrete service classes. In the picture this is represented as if it were accessible through the socket where the functionality of a concrete service must be plugged in.

The ECASrvIntf interface, on the other side, is responsible for the run-time processing of notifications. Notifications of interest are received by the subscriber component and then passed to the service itself by means of the process method defined in this interface. The implementation of this method should process one notification at a time by resolving with its content the attribute references of the WhatToDo object and finally evaluate it. Figure 6.10 shows a subscriber receiving a notification (1) and how it is passed to the service through the ECASrvIntf interface (2). The notification reaches the process method implementation (3) where it is processed. If it is the case, the notification is simply re-published (4) through the publisher which was accordingly

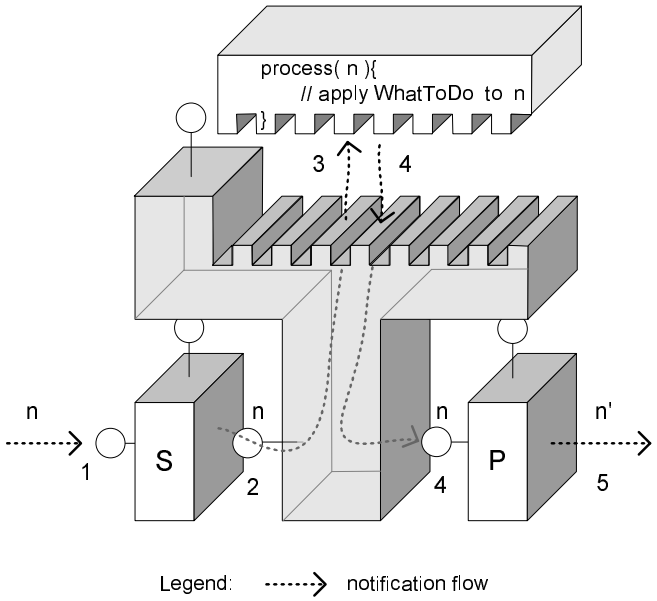

Figure 6.10: ECASrvIntf's process method in context

configured during the registration phase. The publisher passes the notification to the underlying communication mechanism (5).

The class organization presented here provides two main benefits. The first one is that only one class encapsulates platform-specific code (functional logic), making the implementation of elementary services independent of the service platform. The second is that the development of an elementary service concentrates simply on the implementation of the ECASrvIntf, that means rewriting the process method.

In the following sections the implementation of concrete elementary services are presented.

6.3.3 Condition and Filter Services

The condition evaluation service is responsible for evaluating the condition part of rule definitions. Thanks to the class organization presented in the previous section, a condition service implementation must only extend the ECAElemSrv class by re-writing the process method. Figure 6.11 presents a schematic view of a condition service that has a publisher and subscriber already attached.

Basically this kind of service implements an interpreter of expressions. What is going to be interpreted is contained in the WhatToDo variable which also includes references

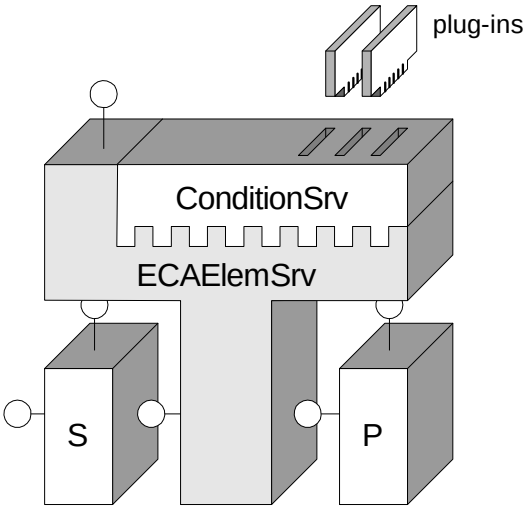

Figure 6.11: Schematic view of a condition service

to notification attributes.

In addition to simple boolean predicates, expressions here can include a combination of instructions like, remote method invocations, queries on databases or another kind of connection with external systems. In order to interpret them, two alternatives are given. In the first alternative the interpreter is built in the service itself, and in the second a plug-in approach is used. To facilitate the interaction with diverse external systems in a flexible way, the plug-in approach is followed in this implementation. The set of instructions that can be processed on external systems is defined by using the ontology. By means of these concepts the corresponding plug-in interfaces are defined and expressions can be instantiated. More details about plug-ins can be found in Section 6.7.

As mentioned above, expressions can contain instructions with arguments that refer to notification attributes. Because instructions are an integral part of the ontology, they can also associate contextual information to them. This feature is useful in order to resolve integration problems with heterogeneous systems and it allows a high-level (domain-specific) description of expressions. For instance, in a simple distance comparison the "metric system"can be defined as context, and automatically distances expressed in different systems are converted to the metric system in order to correctly interpret them.

Algorithm 6.1 presents pseudo code that shows at high-level how notifications are processed. Notice that incoming notifications are processed one at a time, where references to notification attributes are first resolved and then automatically converted to

the target context contained in the instruction if needed. Next, the expression must be evaluated (eval method) and if true, the notification is republished in order to reach the next service in the chain of rule processing services.

Algorithm 6.1 Notification processing - process method

Input: `notification`: an incoming notification of interest

Purpose: evaluation of notifications according to WhatToDo expression

```
vars ← get list of attribute references from WhatToDo
values ← obtain values of vars from notification
WhatToDo' ← resolve WhatToDo with values
if this.eval( WhatToDo') then
  p.publish( notification)
else
  discard notification
end if
```

The eval method implements the evaluation of condition expressions. Essentially this method separates the whole expression into atomic predicates that form an evaluation tree. Each of these predicates is analyzed in order to delegate its interpretation/execution to the corresponding plug-in. The returning results are then replaced in the evaluation tree which is finally evaluated.

The functional behavior of this service is simply inherited from the superclass, which essentially is responsible for subscribing notification of interest, configuring the publisher and setting other parameters like the expression to be interpreted (contained in WhatToDo).

6.3.3.1 Filter Service

As explained in previous chapters, a filter service can be seen as a specialization of a condition service. Both take incoming notifications and verify a boolean predicate, selecting notifications by discarding those that do not evaluate to true. However, filters are restricted to boolean predicates that involve only attributes contained within notifications. This makes their implementation simpler because they do not have external connection with other systems in order to evaluate predicates.

The eval method presented in algorithm 6.1 can be implemented considering boolean connectors and comparison operations, like equal, less than, less or equal than, etc. Notice that, contextual information may be used to correctly compare data.

6.3.4 Action Execution Service

The action service is in charge of executing the action part of rule definitions. This can include (a sequence of) instructions that interact with external systems like, operations on a database, execution of procedures/methods on external applications, etc. But the number and kind of external systems is not fixed and it probably depends on the application domain where the rules are used. For this reason, a flexible approach is required. As explained above and as it will be detailed in Section 6.7, plug-ins are used in this implementation to flexibly delegate the execution of instructions to external systems.

The action service is characterized as the last service (end/final consumer) in the rule processing chain. Consequently, such services have only attached a Subscriber component (as depicted in Figure 6.12) and no notifications related with the rule processing are required to be published.

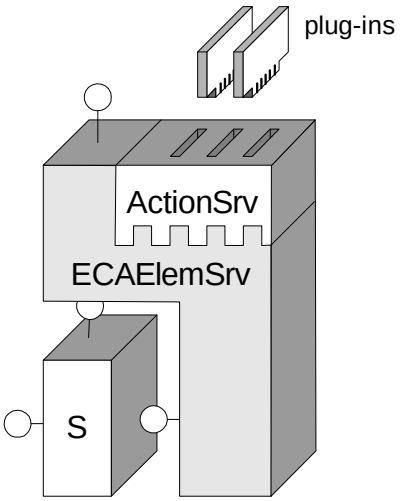

Figure 6.12: Schematic view of an action service

The implementation of the process method is similar to that described in Algorithm 6.1. Here the eval method processes one instruction at a time from the set of instructions. Each instruction is analyzed in order to find the right plug-in where its execution is delegated. Plug-ins can be loaded dynamically if needed. Pseudo code of the eval method implementation is shown in Algorithm 6.2.

Algorithm 6.2 Action execution - eval method

Input: `WhatToDo`: a sequence of instructions (references to notification attributes are already resolved)

Purpose: Delegation of instructions to the corresponding plug-in

```
for all instructions i in WhatToDo do
  k ← obtain kind of instruction of i
  pl ← obtain reference to the plug-in for k
  if ( not pl ) then
    ref ← find plug-in in the plug-in registry
    pl ← plug( ref )
  end if
  i' ← convert parameters of i according to pl's context
  pl.execute( i' )
end for
```

6.3.5 Timestamp Service

Two timestamp models were implemented in order to provide a timestamp service: i) the single clock source model, where the clock can be synchronized in order to appropriately reflect real-world time-related happenings; and ii) the accuracy model, where the clocks of the computers involved are synchronized using the Network Time Protocol (NTP) [Mills 1992].

6.3.6 Alarm Service

The alarm service is the source of temporal events (absolute, periodic and relative), being responsible for announcing scheduled events at the right time. This service is configured by passing a temporal event instance that represents the desired temporal happening. The service looks into the instance content to obtain the date and time in question. With this information, it schedules the clock to return at the right time the same temporal event instance (passed as argument) which is then published through the publisher component. This procedure is graphically presented in Figure 6.13.

The description presented above represents only the scheduling of an absolute temporal event. In case of periodic temporal events, the clock is configured appropriately in order to announce this event repeatedly according to its periodicity.

Scheduling relative events is a little bit more complicated. This kind of events contains an event of reference and an amount of time. Informally, when the event of reference

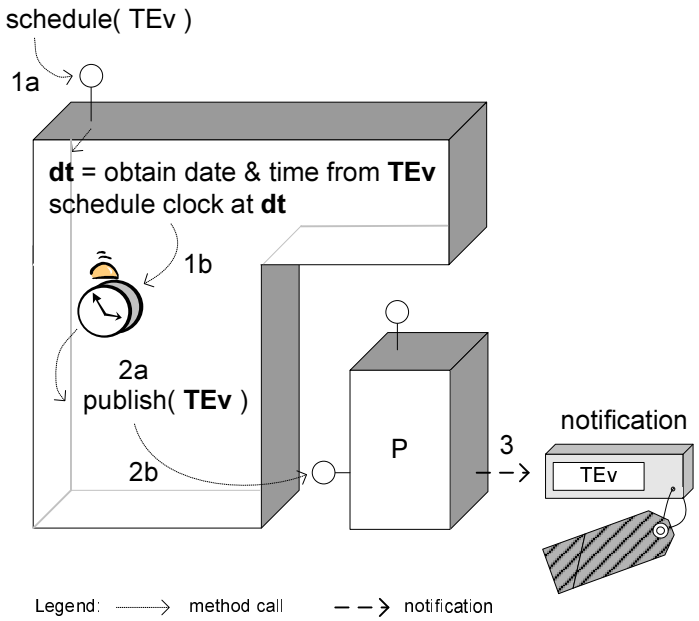

Figure 6.13: Alarm service. Operations involved in the scheduling of an absolute temporal event

occurs the clock must be scheduled adding the defined amount of time to the occurrence time of the event of reference. Consequently in order to support relative events, the alarm service requires now a subscriber component in order to be able to subscribe to the events of reference. Once such an event is received, its occurrence time is obtained and the corresponding amount of time is added to it. As a result, an absolute time is obtained and with it the clock is scheduled in order to announce the relative event.

6.3.7 Repository Service

The repository service is responsible for maintaining all related information about the active functionality service. In particular, the repository stores the following information:

- subject name space organization (depth of control and event content parts, names in use, etc.);
- deployed rules, their definitions and which services are responsible for processing them.

- deployed event adapters (their related concept names, their default contexts, and publishing information);

- plug-ins, their interfaces, the set of instructions they understand and references to their code.

All this is described by means of the ontology, specifically by using concept instances that are in fact instances of Java classes. For this reason, transparent persistence of Java objects was used. In particular, FastObjects j1 [FastObjects] was selected because it supports the Java Data Objects (JDO) interface (for storing plain Java objects persistently in data stores) and it provides a small footprint.

6.3.8 Ontology Service

As described at the beginning of this chapter, ontology concepts are specified using Java classes. Concepts are given as pre-compiled Java classes that can be downloaded and used by services, applications, connectors and other clients. The Ontology Service stores and manages concepts and packages. The implementation of the ontology service is realized on the basis of a web server. The web server maintains all ontology concepts organized in packages and responds to concept requests with the appropriate class [Bornhövd and Buchmann 1999]. Additionally the ontology service provides the possibility to obtain information about concepts, like their relationships with other concepts.

The CSF platform offers the possibility to specify a class loader with each service manager who is responsible for a service. In this way, ECA services take advantage of this possibility and they use a class loader which is responsible to contact the ontology service when needed.

From the client perspective, ontology concepts are used in two forms: static and dynamic. The static alternative is characterized by those concepts that form part of the infrastructure and are used for the implementation of services. In this way, concepts can be statically attached/included within the bundle of service files. On the other hand, the dynamic use of concepts, as its name indicates, depends on the data/events that are being exchanged. If the PlaceBid event is delivered to a set of consumers, then those consumers need to dynamically load the PlaceBid concept definition from the ontology service.

6.4 Event Adapters

Event adapters convert source specific events into concepts from the domain-specific vocabulary and add proper context information in order to support their correct interpretation.

6.4.1 Application Adapter

The adapter facility is built on top of the Ontology API. This software makes it possible for typical adapter tasks to be simply configured reducing development effort. These artifacts are coupled to an application that generates events (or exchanges data). The adapter is configured by defining all the concepts that a particular application exchanges together with their default contexts. Here the default contexts of each concept explicitly represent the implicit design assumptions of corresponding data generated by this particular application. For instance, the default context of Prices for a particular application in the USA can be defined as being "US Dollars", and similarly other default contexts must be defined in order to attach the right metadata with the data exchanged. Of course, default values can be overwritten by the application if necessary.

The implementation of the adapter facility uses the Ontology API to instantiate ontology concepts, and a publisher component to deliver exchanged data through the semantic pipeline. Figure 6.14 shows a high-level graphical representation of an adapter facility in context.

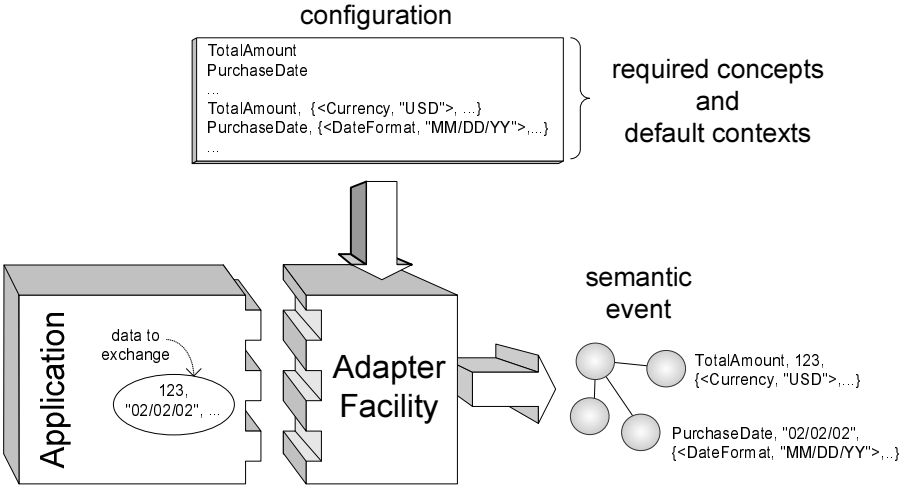

Figure 6.14: Adapter facility in context

6.4.2 XML Adapter

This component is responsible for mapping XML documents into semantic objects (concepts). The current implementation was partially re-written and extended from that presented in [Kottig 2000]. The main extensions are related to process notifications, that arrive in form of strings (XML-conform) and are then mapped into semantic objects.

The mapping rules are not coded in the adapter itself but defined in a mapping file that is used to configure the adapter. The mapping file is organized in two parts: i) how elements and attributes are mapped into concept classes and ii) their corresponding semantic context specification.

The mapping process involves two phases. The first pre-processes the configuration information. Here the DTD of the XML-conform input is extended with the information contained in the mapping file, obtaining as a result a new DTD. This new extended DTD contains associated to XML elements and attributes the name of their corresponding concept classes and their contextual information.

During the second phase (at run-time) XML documents are parsed using the extended DTD obtaining a DOM tree that contains additional information (mapping info). After that, the DOM representation is traversed while instantiating by reflection the corresponding semantic objects. Figure 6.15 presents an abstract view of the XML adapter together with the mapping file and it shows how XML documents are transformed into concepts.

6.4.3 Converting Semantic Objects into XML Documents

Semantic objects can be exported in the form of XML documents. This task is carried out by traversing the semantic object structure of a concept instance generating a DOM representation. Context information is transformed into an additional XML element or into attributes of the corresponding XML element. Finally, the DOM representation is exported into an XML document.

6.5 Rule Definition

As described in Section 3.2 these are two perspectives that must be distinguished regarding ECA rule representation: how rules are expressed by the end-users and how they are represented inside the system. With this in mind the representation of rules

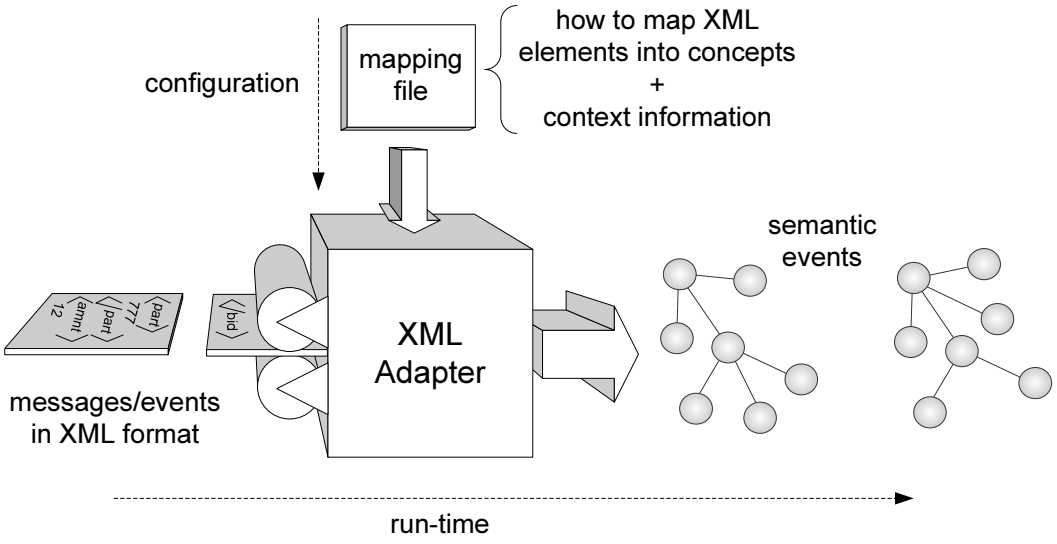

Figure 6.15: XML adapter

was organized into three layers. In this section the attention is concentrated on the first two layers, the external and the conceptual. In this implementation the third layer (internal) maintains the same representation as that used in the second layer. With this organization the active functionality service provides the possibility to offer end-users a variety of ways to define rules.

The external representation is related to the question how end-users specify ECA-rules. For this purpose, the requirements of the user and the application with respect to usability aspects must be investigated. According to the resulting criteria the interface for defining rules must be created. As a result, for instance, a variety of interfaces for the same domain can be offered taking into consideration different types of users. This could be the case when using a textual definition for experts and simple form for non-technical users.

This flexible approach is founded on an intermediate, ontology-based representation of rules where rule-related concepts must be defined and specialized if needed. Rule definitions are then represented by instantiating such concepts. As mentioned at the beginning of this chapter an Ontology API is provided in order to facilitate the instantiation of concepts. However, in order to map the external representation into concept instances a transformation step is needed. This transformation depends on the means offered to the user, for instance, a rule compiler that according to a rule definition language creates corresponding concept instances, or JSPs, Servlets, or Cocoon components that are in charge of transforming rules defined by means of web forms. This approach is depicted in Figure 6.16.

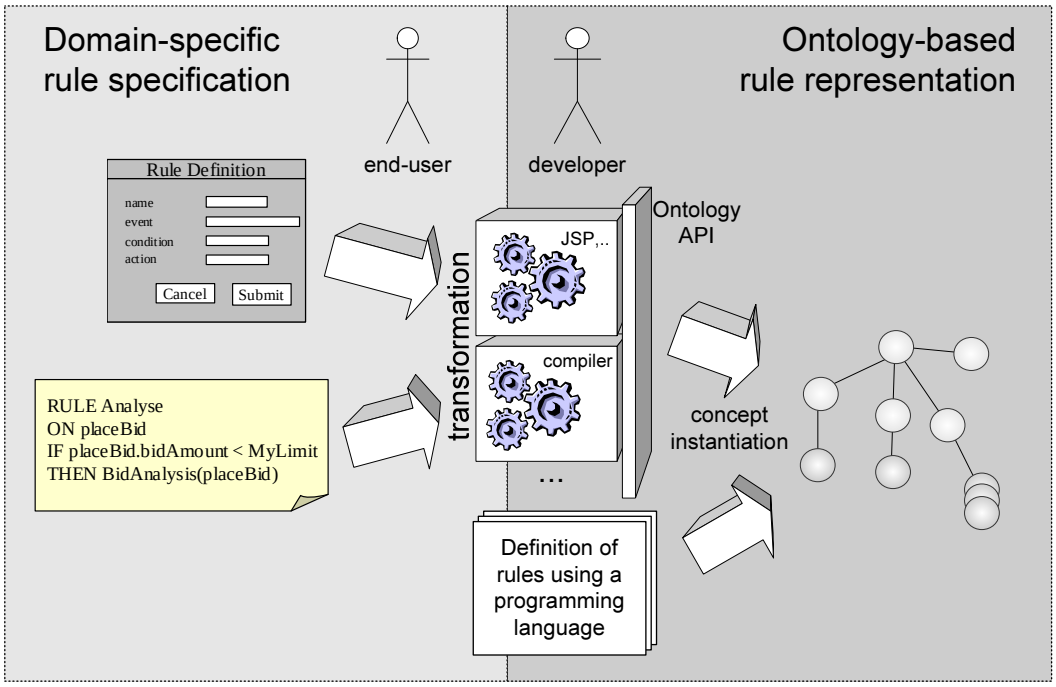

Figure 6.16: Rule definition approach

The transformation process provides the possibility to hide or expose technical details about rule properties (e.g. consumption or coupling modes) to the end-user.

There is also a low level alternative where rules can be defined by writing a program that instantiates rule concepts with the appropriate content.

Summarizing, the approach followed here allows a flexible definition of rules that can be customized for different domains and according to the user's knowledge. It is important to highlight that the same underlying active functionality mechanism is used in all scenarios. Once rules are defined, they are transformed into ontology concept instances and these are then passed to the ECA-Manager that is explained in the following section.

6.6 ECA-Rule Manager

The ECA-Rule manager is the representative of the active functionality service. It is responsible for the registration of rules and for determining which elementary services are in charge of their execution. That means, it is responsible for building the rule processing chain by composing services.

6.6.1 Building the Rule Processing Chain

Figure 6.17 shows the ECA-Rule manager in its architectural context including the steps involved in registering a rule. The manager receives a rule definition in the form of a concept instance for registration (1), it analyzes it, and obtains the (three) main parts (i.e. Event, Condition and Action) (2). The event part can contain a primitive or a complex event. The condition and action parts can include an expression and a sequence of instructions respectively that need to be interpreted/executed at each elementary service. In particular, conditions are examined with the purpose of finding intra-notification predicates that could be transformed into filters. Notice that not all rule definitions must include the three parts.

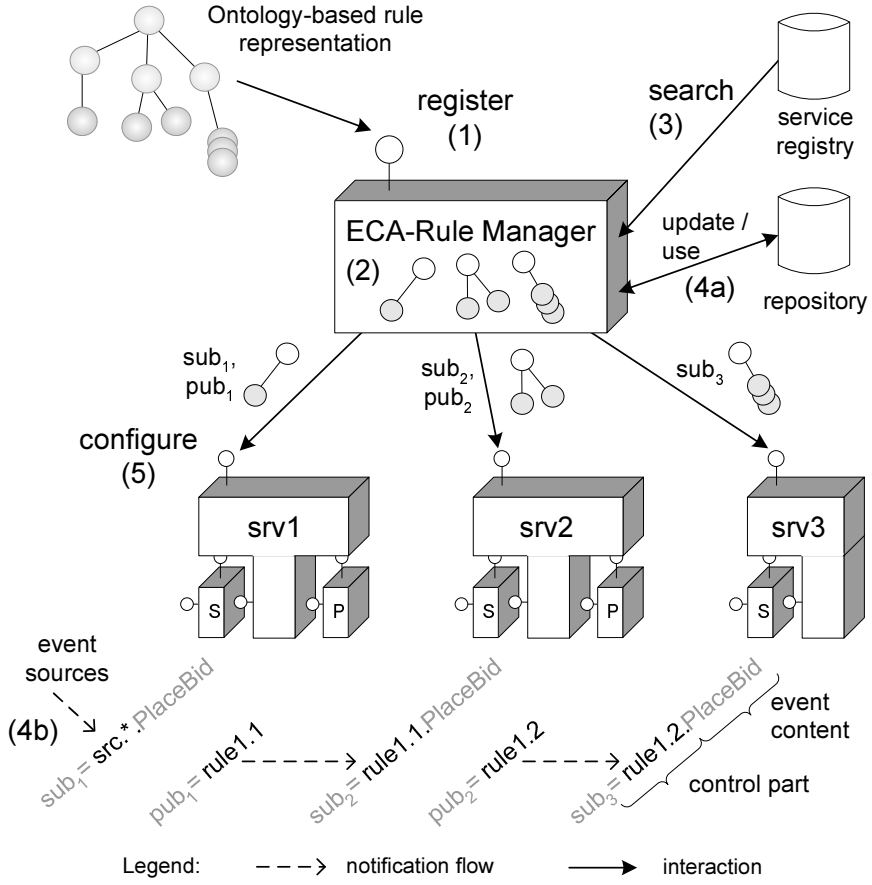

Figure 6.17: ECA-Rule Manager in context

The ECA-Rule manager searches the service registry for elementary services that can process these parts (3). Notice that at deployment time services have registered not

only their name, interface and version number but also other characteristics and properties like, where the service is located, what kind of sentences they can process, the quality of services they offer, etc. Consequently, the search criteria can be more precise by including these characteristics.

Once the candidate services are found, the processing chain is sketched by finding the proper subject name, particularly the control part (4a). In the case presented in the figure, the depth of control part of the subject is set to two, and its content slot includes the names of the services (as depicted at the bottom of the figure). The first position of the control part represents the RuleId while the second one is used to explicitly put the sequence of execution of involved elementary service. In a similar way, the depth of the event content part of the subject is set to one, which means that it only includes the concept name of the event that fires the rule. Obviously, the deeper the event content part the more precise can the subscription be (e.g. by including identifying attributes as explained previously in this chapter).

Taking into account all this, for each elementary service that participates by processing the submitted rule, a subscription pattern (sub) and the publisher's control subject information (pub) are generated (4b). Together with this information and with the corresponding parts obtained in step 2, the selected elementary services are configured (5) in order to participate in the rule processing chain.

Notice that the manager is the only component that has a high-level view of the processing chain, knowing which elementary services participate in the process of a particular rule. Moreover, an elementary service does not know from where notifications come and which are the consumers of the notifications it publishes.

6.6.2 Rule Selection Policy

As mentioned previously, the prototype implements no rule selection policy. That means, that multiple rules with the same triggering event are executed in parallel. In this case, the rules directly subscribe to the triggering event. Remember that events are disseminated using a pub/sub mechanism. Thus, when the event in question is notified the first elementary service of all corresponding processing chains gets the notification and begins with the process in parallel.

For instance, consider the following example that includes three rules. The definition of the first two rules contains a simple event, a condition and an action part. The third rule is defined with a simple event and an action. Notice that the event in question is the same for all three rules (PlaceBid). Figure 6.18 shows the processing chains of these rules. Here the first elementary service of the processing chain of each rule subscribes

directly to the event in question (src.*.PlaceBid). Thus, when the event is published all three rules are automatically fired in parallel.

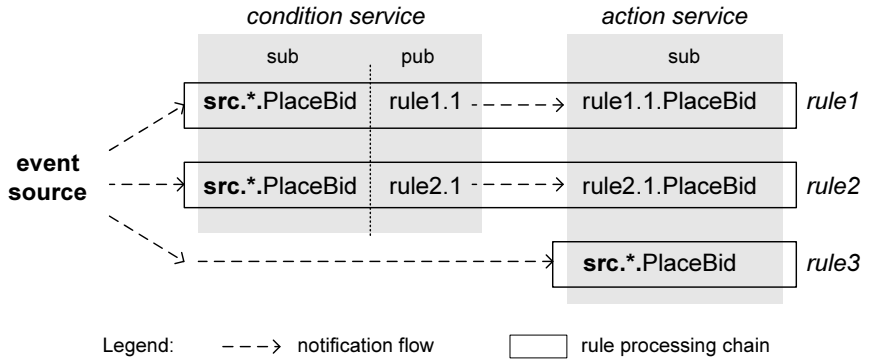

Figure 6.18: Rule execution without a rule selection policy

Clearly, other rule selection policies can be used by simply putting a rule selector service in front of the rule processing chain if they share the same triggering event. This selector is in charge of subscribing to the event in question. Thus, when the event is propagated, it must resolve which rule should be fired by applying the appropriate criteria (random, priority, etc.). Once the decision is taken, it (re)publishes the notification in the corresponding rule processing chain. Following the example presented above, Figure 6.19 shows the same set of rules but now placing the rule selector in front of the processing chains. The selector is then responsible for choosing a rule and finally it (re)publishes the event in order to reach the selected rule processing chain.

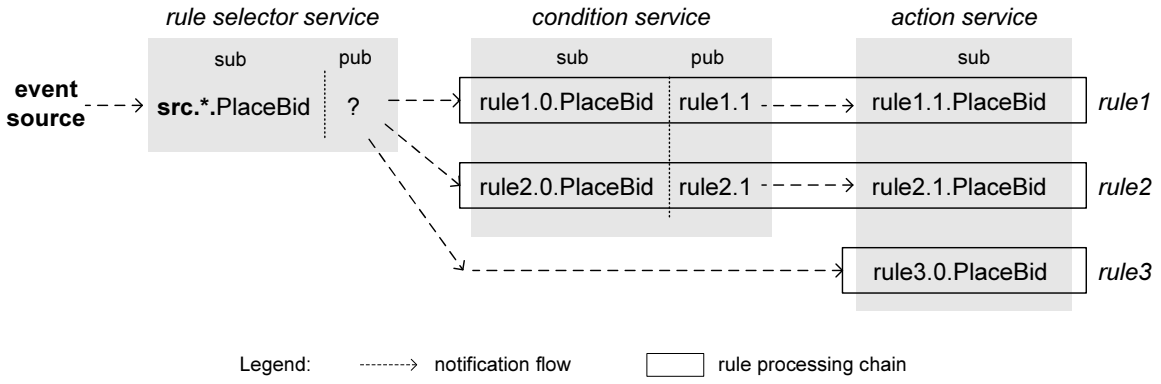

Figure 6.19: Rule execution with selection policy

Summarizing these five steps, the ECA-Rule manager is responsible for registering rules. This is carried out by composing elementary services according to the rule

definition and the rule processing model with the purpose of building a rule processing chain instance which at run-time will process the rule in question.

6.7 Plug-ins

Similar to the principles used in today's browser technology, plug-ins extend the capabilities of a service in a specific way. They are responsible for delegating the execution of instructions to the system they represent. In other words, they play the role of a proxy of an external system providing an interface that includes a set of methods that are effectively executed there.

In this work, plug-ins can be classified in two groups according to the kind of system they represent. *Application-specific* plug-ins provide functionality to access a particular application, e.g. a calendar application. *Generic* plug-ins, in contrast, offer functionality to access base software like databases, workflow engines, etc.

Following the ontology-based infrastructure approach, the methods defined in a plug-in interface have their corresponding concept definitions in the ontology. These concepts are used as instructions (of conditions or actions) in rule definitions. In this way, instructions and plug-in interfaces are defined using a common vocabulary, allowing an incremental and dynamic set of instructions according to the domain in question. Thus, interactions with new systems can be added by adding the corresponding concepts in the ontology and by developing and registering the corresponding plug-in with the registry.

Plug-ins can be plugged in to a service at run-time. Services maintain a list of methods provided by the interfaces of the already attached plug-ins. If necessary plug-ins can be loaded on demand by looking up in the plug-in registry and loading the corresponding code.

Consequently, plug-ins offer the possibility to interact with diverse systems making the instruction set that can be included in rule definitions more flexible and powerful.

Additionally, and considering that application-specific plug-ins are the connection with external applications, and that these applications probably have their own design assumption, it is necessary to put the data in the right context before it is exported. Consequently, and in order to avoid misinterpretations, plug-ins must define their contextual information (design assumption) so that data passed as arguments can be transformed before the instruction is executed. This kind of feature is easily implementable due to the use of conversion functions supported by the underlying ontology infrastructure.

Within the scope of this prototype some plug-ins were developed which are used in the scenarios presented in the next chapter.

For instance, a simple Workflow plug-in was developed to allow rule actions to start workflow process instances. In particular, it implements the interaction with the HP Process Manager v5, which is the workflow engine of HP. As mentioned above, concepts related to workflows (in general), like WorkflowProcess, WorkflowItem, WorkflowEngine need to be defined in the ontology as well as the instructions that can be used from the rule perspective (e.g. startWorkflowProcess). The plug-in implementation is responsible for transforming the content of the concept instances passed as arguments and for putting them in correspondence with the Process Manager API.

6.8 Summary

This chapter presented details about the implementation of the prototype. Since it was built using a service framework the framework's main characteristics were presented as well as the details about the implementation of ontology concepts. Elementary services were described and it was shown that on the basis of these services the active functionality service can be assembled. It must be noticed that the infrastructure uses the ontology as a common interpretation basis, so independently developed components can be integrated without the hazard of misinterpretations among components. The ECA-Manager is responsible for the composition of elementary services. To define and represent rules a flexible approach was taken allowing customized definition of rules for different domains but represented by means of ontology concepts and exploiting the use of context information. Finally, adapters and plug-ins were described. Adapters "import"events/data from external systems into the active service while Plug-ins "export"data to external systems. Both have the solely goal of maintaining exchanged data semantic intact.

Chapter 7

Using the Active Functionality Service

This chapter explores the main features of the active functionality service developed in this thesis with the help of two different scenarios. In the first section the online auction context is analyzed and a Meta-Auction service is proposed. It provides a unified view of different auction houses and services for category browsing, item search, auction participation and auction tracking. Under these circumstances, a semantically meaningful information exchange among participants is required. Auction-related events are considered first-class information. For this purpose and in order to make these happenings available to all interested participants, the active functionality service is used. Moreover, bidder agents can be defined in terms of rules to automatically react to auction happenings on behalf of their bidders.

The second scenario shows how personalization of vehicles can be realized by exploiting the active functionality service. In this case, it is possible to provide location-based recommendations for repairs or fuel, personalized vehicle settings, and navigation aids to name just a few. This not only provides the capacity of delivering personalized services, but the possibility of applying these settings (or at least parts of them) to different vehicles. This is achieved by using ECA-Rules that react to situations of interest. This scenario explores the interaction with diverse external services using context information in order to exchange the data in the right form; flexible rule execution where the service can be located together with the car computer or distributed in other machines; the integration of events from diverse sources; and flexible rule definition.

7.1 Online Auctions

Auctions are a popular trading mechanism when multiple buyers compete for scarce resources. The advent of auction sites on the Internet, such as eBay or Yahoo! has popularized the auction paradigm and has made it accessible to a broad public that can trade practically anything in a consumer to consumer interaction. The mechanism has become so popular that many e-businesses are using auction mechanisms to handle prices.

Tracking the objects that are auctioned is time consuming. Therefore, some form of notification mechanism is needed to alert a potential buyer when an item of interest comes on the market. Serious art collectors have used similar services for centuries. Agents or gallery owners notify a potential buyer whenever an article that might interest a customer becomes available. In the world of Internet-auctions collectors would like to enjoy a similar service. In addition, a collector might prefer to deal with one common auction portal instead of registering her interests with multiple auction sites. Therefore, the notion of a meta-auction was introduced.

7.1.1 Meta-Auctions

A meta-auction [Bornhövd et al. 2000] allows a potential buyer to roam automatically and seamlessly across auction sites for auctions and items of interest. To realize the meta-auction, several problems must be solved. User-initiated communications lies at the heart of today's auction systems, which are therefore not appropriate for this kind of applications. Moreover, they will not scale properly. The large number of interconnected users and systems, as well as their wide-area distribution imposes particular restrictions with respect to response time and network bandwidth. Internet-scale information systems therefore must leverage proactive information dissemination and caching techniques. However, typical client/server and n-tier system architectures are merely based on a request/response interaction and do not take into account the asymmetric nature of such systems [Acharya et al. 1995], where meaningful data flow is from the backend-tier to end-users.

Furthermore, the query metaphor from the database domain is currently the primary means for information acquisition, which results in the user polling for changes and happenings of interest. Notifications about events, such as the placement of a highest bid, and their timely delivery to the user represent valuable information. Therefore, publish/subscribe as an additional interaction paradigm is needed to make the efficient dissemination of process-related information possible.

Each site participating in the meta-auction system provides information about items and the auction process but does not share a global data schema nor a global schema for notifications. Still, all participants come from the same application domain and at least conceptually, share a common vocabulary. While in most of today's systems the vocabulary is left implicit, an ontology-based infrastructure for explicit metadata-management is proposed on top of which the meta-auction service can be realized. The suggested ontology-based infrastructure provides common vocabularies for semantically meaningful exchange of data and notifications, and supports incremental integration of participating information systems as needed.

Consider the case of a collector. With the current auction sites, she has to manually search for the item of interest, possibly visiting more than one auction site. If successful, she might end up being engaged in different auctions at multiple auction sites. There are two obvious shortcomings to this approach: first, the user must poll for new information and might miss the window of opportunity, and second, the user must handle different auction sites with different category setups and different handlings. This motivates the need for the meta-auction broker, which provides a unified view of different auction sites and services for category browsing, item search, auction participation and auction tracking.

Events that arise in the context of an auction process should be treated as first-class information and propagated as notifications to the users who subscribed to the event. Figure 7.1 shows a classification of auction-related events. Propagation of events leads to a useful and efficient non-polling realization of an auction tracking service.

In this scenario, it is mandatory to cope with heterogeneity. Today, the exact meaning of terms, entities and notifications used by different auction sites is still left implicit. To enable the brokering between different participating auction sites, the precise understanding of the terms used by each site is needed and should be made explicit through a domain-specific common vocabulary. This is a prerequisite for semantically meaningful information exchange between a frequently changing set of independent participants in a large-scale business scenario.

As mentioned before, adapters resolve heterogeneities with regard to organization and structure of the data, and the use of different terms referring to the same real-world aspects. In addition, metadata is added to the available data to make implicit modeling assumptions concerning organization and meaning explicit. Based on this representation, heterogeneities in the semantics of the data, e.g. use of different units of measure, scale factors, derivation formulas, coding, or naming schemas, can be resolved by the adapters at "integration"time.

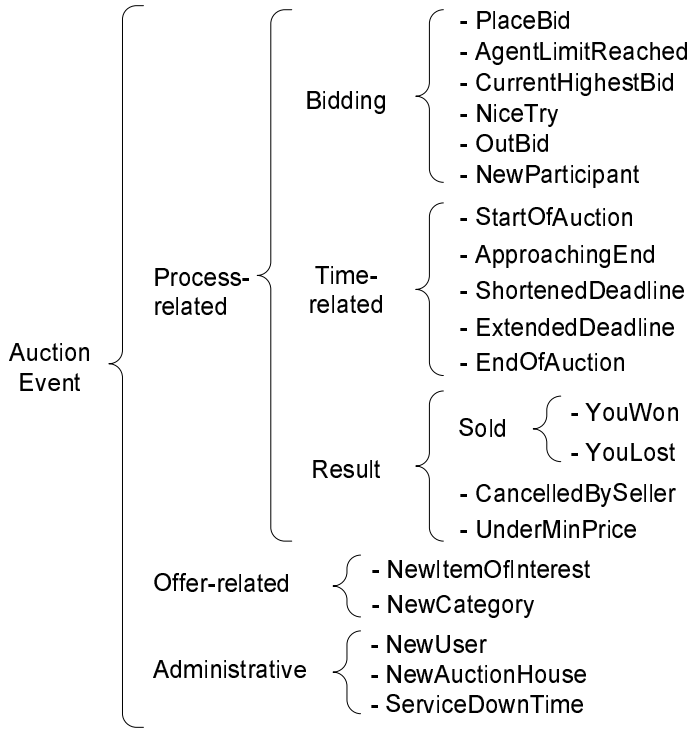

Figure 7.1: A classification of auction-related events

7.1.2 Auction Service

The auction process itself can be defined using statecharts [Benyoucef and Keller 2000] and because they are event-driven, they can be easily implemented with ECA-rules. In this way, different sets of rules can describe different types of auction processes (ascending, reverse, dutch, etc.). Figure 7.2 shows a statechart of a simple ascending auction process.

The definition of a statechart can include contextual information which helps to provide a high-level definition of the auction process. For instance, in this particular scenario where auction participants could be around the globe, they can bid using their own currency. Here comparisons among BidAmounts are specified at high-level.
This kind of statechart definition enables the derivation of high-level ECA-rules where any conversion (if necessary) is carried by the underlying active functionality service.

Derived rules include as part of the action the announcement of state changes on a state chart. Consequently, they automatically publish auction related events. Under these circumstances, every state change produces an event notification, making the auction process happenings (e.g. NewHighestBid, UnderMinPrice, Sold, etc.) available

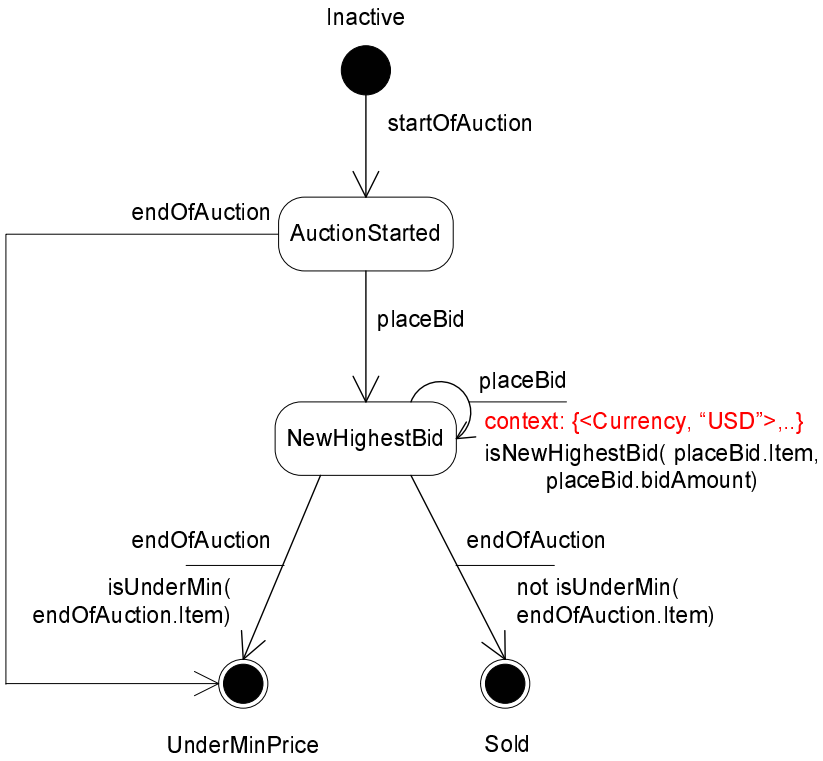

Figure 7.2: Statechart of a simple ascending auction process

to interested participants. These events are disseminated using the notification service proposed in this thesis. Thanks to the concept-based addressing, publishers and subscribers use a semantic level of subscription which is common to all of them.

Figure 7.3 shows a rule that corresponds to the transition that reacts to the placement of a bid of a participant (placeBid) moving from the NewHighestBid state and back to the same state. This state change is only made if the incoming placeBid has the highest bid amount. If so, it generates the corresponding notification. As shown in the figure, the boolean expression associated with this transition has attached a context definition which allows a correct comparison of bid amounts.

The maintenance of the state of all auction instances is performed by the *auctionProcess* that implements boolean predicates (e.g. isCurrState, isNewHighestBid, isUnderMin) and other methods (e.g. changeStateTo, publishNewHighestBid).

After rules are derived they are passed to the ECA-Rule Manager that in turn performs the following steps: i) it breaks derived rules down into elementary elements (e.g. event, condition, action), ii) it searches for the corresponding services, and iii) selected services are configured with these elementary elements. At run-time the elementary services

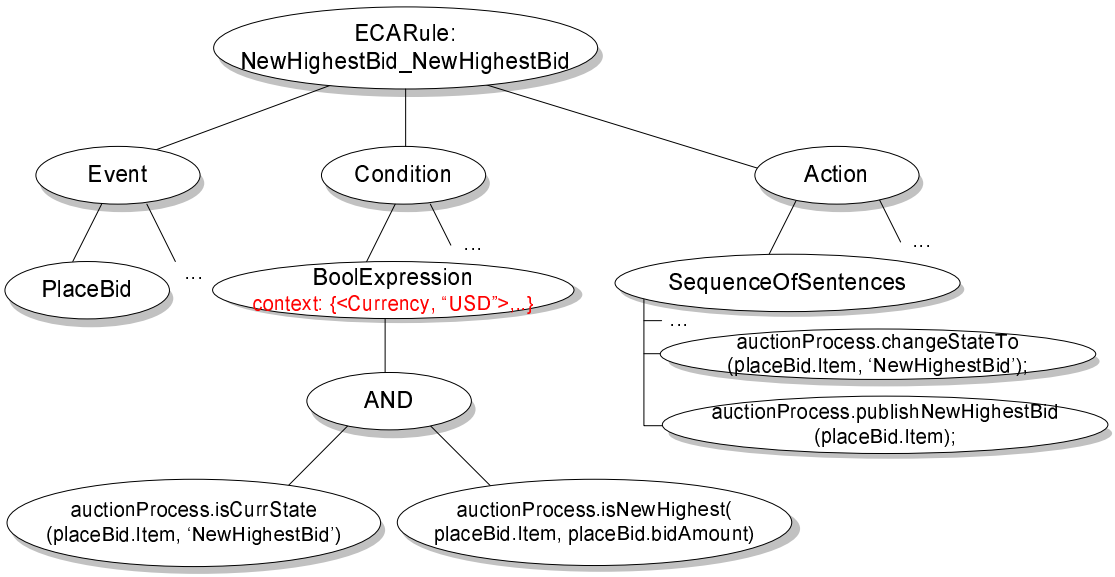

Figure 7.3: Graphical representation of an ontology-based rule that is derived from the auction statechart

involved are responsible for executing several auction processes concurrently.

7.1.3 Bidder Agent

To track an item of interest during an auction process an agent can be used, for example, to ascertain that another bidder has reached a highest bid, or that the deadline of an auction is approaching. Here bidders can benefit from a rule service to program their own agents. In contrast to current agent bidders, where they are owned, controlled and implemented by the auction house, these agents can react to happenings of the auction process according to the bidders' strategy and can be located in "any"computer on the network.

7.1.4 Comments & Conclusions

As it was presented above, this scenario benefits from using the active functionality service. In particular, it shows how to profit from the underlying infrastructure that enables the integration of heterogeneous information from multiple, independent sources. For this scenario, a domain-specific ontology (Auctions) was defined where related online auction concepts were specified (See Appendix C.1). This allows a clear separation

of domain-specific terminology and those terms used by the active functionality infrastructure. Additionally, auction-related events are represented using the corresponding ontology concepts and augmented with meta-data that describes the assumptions of the data at each source.

Notice that auction-related events are considered first-class information and they are pro-actively disseminated to all participants by means of a concept-based (publish/-subscribe) notification service (which is part of the underlying infrastructure).

The auction process itself is described using statecharts where contextual information can be included in order to maintain rule definition at a higher level of abstraction. From this definition ECA-rules are derived where the publication of process-related events are included. Additionally, bidders benefit from the use of the active service by capturing their bidding strategy using (ECA-)rules.

The auction service proposed here (that supports not only ascending but also reverse, dutch and other kinds of auctions) can be reused in other trading scenarios like B2B, marketplaces, etc.

In the next section the same underlying active functionality service will be used in a different scenario, namely to personalize vehicles according to drivers' preferences.

7.2 Rule-based Vehicle Personalization

Similar to other pervasive computing environments, cars will see a convergence of Internet, multimedia, wireless connectivity, consumer devices, and automotive electronics [Hansmann et al. 2001]. Assuming this context, wireless links between car systems and the outside world open up a wide range of telematics applications. Automotive systems are no longer limited to information located on-board, but can benefit from a remote network and service infrastructure.

Consider for instance an automobile scenario, where vehicles, persons and devices have a web presence (or portal). Within this scenario new possibilities emerge, for example the adjustment of instruments according to personal settings stored in the portals. This not only provides the possibility of adjusting instruments of a vehicle, but the chance of applying these settings (or at least part of them) to different cars.

Under these circumstances, a frequent traveller can use any rented car and it automatically adjusts its instruments (display of units of measurement, radio stations, vehicle's internal temperature, seat settings, etc.) according to the driver's preferences. But not only instruments can be adjusted, services can be personalized too. Services such as,

"find and set the route to the next gas station", or "book an appointment to change oil"can take into account vehicle manufacturer's, company's, or driver's preferences.

Personalization should be mostly invisible and automatic, customizing the user experience according to her/his preferences. Under these circumstances, the use of an ECA-rule service seems to be appropriate. Rule-based personalization uses specific information about individuals to react in a certain situation according to their preferences [Lopes de Oliveira et al. 1997].

7.2.1 Scenario-related Technology

7.2.1.1 The CoolTown Model

Under the CoolTown model [Kindberg et al. 2000], people, places and things have a "web presence", that extends the "home page" concept to include all physical entities and to include automatic system-supported correlation of the home page or *point of web presence* with the physical entity (see Figure 7.4). This web presence provides current information and services relevant to its representative. This model supports nomadic users, based on the convergence of web technology, wireless networks and portable devices.

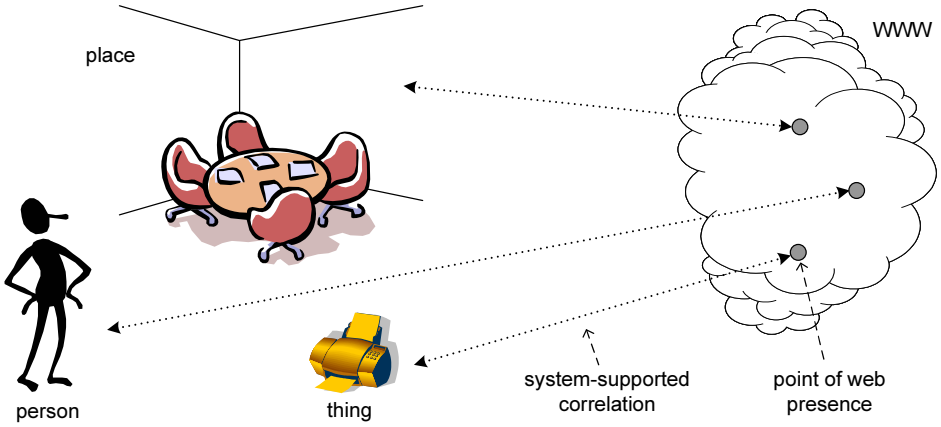

Figure 7.4: CoolTown model

As mentioned before, physical entities are divided into three categories. *People* are the users of *things* and the occupants or visitors in *places*. Places have a special role as the container for people and things. Things become web-present by embedding web-servers in them or by hosting their web-presence within a web-server.

7.2.1.2 Portals

A Portal provides information and services relevant to its representative. For instance, a device can be optimized for tasks related to using, managing, or enhancing it. The portal takes advantage of functionality and information available on the device and adds the manufacturers' content and services. Furthermore, it allows additional services to be added. For example, in a vehicle context, an "Auto-biographer"service can offer an automated history of the events related to the vehicle, from the day it rolls off the assembly line.

Portals should reflect not only static but also dynamic relationships to other portals. For example, the portal of a vehicle can include the static relationships to the radio and to the onboard navigation system that are built-in in the vehicle but also dynamic/spontaneous references to the portals of people and devices that are currently inside the vehicle. The information maintained within a portal can be considered important contextual information.

A portal manager is then responsible for handling all this information. Additionally, it provides restricted access to the contents of a web presence according to the security policy defined and to the visitor's credentials.

7.2.1.3 The Box

In this scenario it is assumed that vehicles are equipped with a GPS receiver and a box. This box plays the role of a mediator between the vehicle itself and the external world. It can access a vehicle's electronic and diagnostic interfaces (like J1850 [Society of Automotive Engineers 1994], ODB-II [Bohacz] or ISO 9141 [ISO 1989]). For instance, it can be used to read or change parameters and measurements of the engine; it can access the state of sensors; it can contain information about the occupants inside the vehicle (including not only persons but also devices); etc. The box is responsible for announcing status changes to its portal, keeping it always up-to-date.

Additionally, the box acts as a proxy of the instruments inside a car. These instruments are modelled as services which provide a service interface. In this way, these instruments can be contacted from the outside in a standard way and the software in the box is responsible for effectively manipulating the instruments.

7.2.1.4 Services

Services are organized in two groups according to where they run. External services are those that run outside the car. For instance, location services (like, where to find

a gas station, or a hotel), news services, weather forecast services, route planning and traffic information services (calculate the optimum route to a specific destination by considering traffic jams, road constructions, and even bad weather conditions), just to name a few.

By contrast, internal services represent those services that run inside the car. For example, navigation services (guide drivers to their destination), geographical positioning services, music player service, text-to-speech (TTS) service, and those related to the instruments of the car. As mentioned in the previous section, these services are accessed through the box.

7.2.2 The Vehicle Scenario

The scenario presented in this section follows the CoolTown model by assuming that persons, things and places (in this case vehicles) have a web presence. Figure 7.5 shows – by means of the dashed arrows – the correlation of real-world objects and their portals. As part of the model, things and persons must be recognized when they get into a vehicle (by distinguishing the key's owner, or another identifying device). This happening of entering a place is conveyed to the place portal. In this particular case, occupants of the vehicle are maintained in the car's portal by keeping the dynamic relationships to them (and vice-versa). This is also depicted in the figure with solid arrows. As mentioned before, portal managers are responsible for administrating a portal's information. Here this functionality is provided by the Web Presence Manager [Caswell and Debaty 2000]. By means of a web browser it is possible to access a portal. Portal settings can be managed according to the security policy defined. By specifying how information is shared, types of information and services can be customized.

The box is responsible for announcing changes in the status of a vehicle in order to reflect current information in their portals. Today, vehicle sensors are detecting all possible parameters, like rainfall, air pressure in the tires, fuel level, state of the engine, door locks, seats, mirrors, engine problems (with different levels of severity), etc. For instance, consider the fuel level sensor. When running out of fuel the vehicle can take care of finding the next gas station, considering current geographical position, and driving direction. Moreover, the search process can also consider driver's preferences (e.g. loyalty programs) or manufacturer's recommendations.

In the prototype that implements this scenario, vehicle diagnostics and other sensor signals are simulated through a control panel. For instance, by means of this panel, the level of fuel or the geographical position of the vehicle can be changed or failures can be simulated.

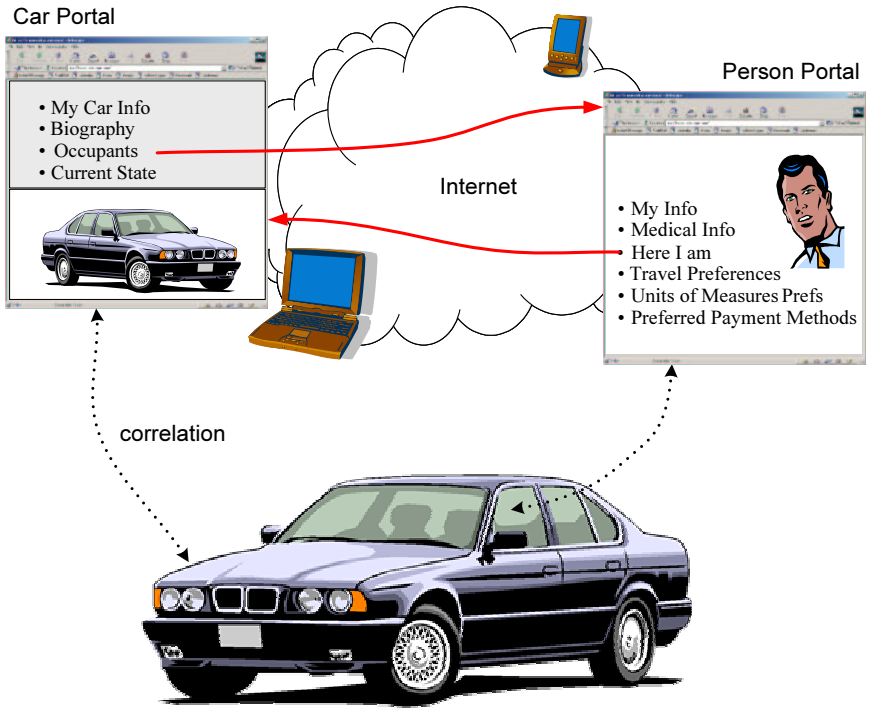

Figure 7.5: Vehicle scenario

The implementation of this scenario includes the definition of required concepts related to the automotive environment. Additionally, some plug-ins were developed to access external systems like the car box, a workflow engine, etc.

7.2.3 Enhancing Portal Managers with ECA-Rules

It is assumed that information and status maintained in portals reflect the current real-world status. In other words portals provide contextual information. Under these circumstances, why not personalize a user's experience considering his or her preferences. Considering that status changes on real-world objects are (always) kept up-to-date on their portals, then reactions to some of these changes can play an important role. With this in mind, portals can be enhanced with ECA-rule capabilities in order to provide customized reactions according to happenings of interest and user preferences. In this way, every time happenings are received by a portal manager (in order to maintain the web presence up-to-date) reactions to them can be carried out.

This enhancement can be implemented by assuming that state changes of real-world objects are disseminated using the publish/subscribe mechanism presented in the pre-

vious chapter. Thus, the portal manager needs to attach a subscriber component. In this particular scenario, car boxes are responsible for announcing vehicle state changes and portal managers subscribe to vehicle events in order to maintain their content up-to-date. In a similar way, car-related rules (specifically the elementary services that are responsible for their execution) subscribe to the events that are associated to their definition.

Figure 7.6 shows a car portal manager in context. Here are depicted the steps involved in firing and executing a rule. Consider the situation where a driver gets into the car and the car instruments are automatically adjusted to her preferences. This situation begins with the driver getting into the car (e.g. by distinguishing the car key) where the box is responsible of its announcement (1). This event reaches the portal manager where the information is updated (2a). Moreover, and as an effect of the publish/subscribe mechanism, the notification is also received by the elementary service that is responsible for processing the corresponding rule (2b). The rule is processed and it consequently reads the driver's personalization information from her portal (3) and contacts the box in order to adjust instruments according to her preferences (4).

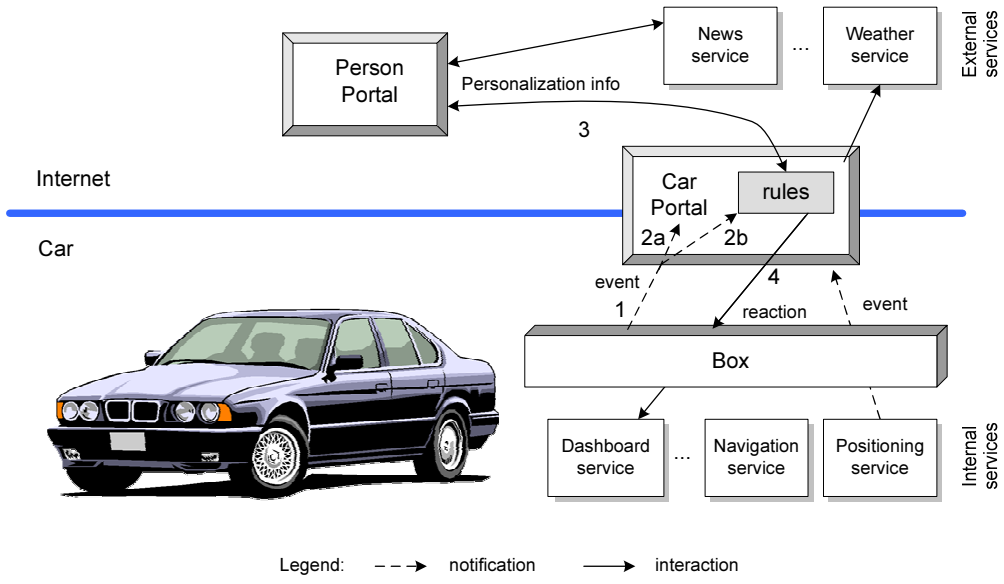

Figure 7.6: Car portal manager in context

The following section provides a set of rules related to this scenario.

7.2.4 Vehicle Personalization using ECA-Rules

Rules in this section are organized in three groups according to the kind of situations to which they are related.

7.2.4.1 Concrete events (sensor signals)

Once the *driver gets into the car*, a rule accesses the driver preferences from the portal and with them vehicle instruments can be adjusted by contacting the car's box. For example, the way instruments display their values to the driver can be customized at least which units of measure should be used (km vs. miles, Celsius vs. Fahrenheit, date/time format, liters vs. gallons, etc.).

When the vehicle is *running out of fuel* a sensor signals this happening. As a reaction, a query to find the next gas station is executed considering current geographical position, current fuel level, destination and driver's (or car's) preferences. Figure 7.7 depicts an abstract definition of such a rule.
Similarly, when a *problem is detected* or a warning of a likely failure is reported an appropriated action can be taken. As a reaction, information can be sent to the car manufacturer, and the closest repair shop according to the driver's (or car's) preferences can be searched. Both rules start workflow processes where the action is effectively executed.

It must be noticed that in the rule depicted in Figure 7.7 the action part contains a context definition which specifies the unit of measure used for volume capacity. This is important since rules interact with external services, in this case a workflow engine. Here, data is exported according to design assumptions that were specified in context of the rule's action. Consequently, the data contained in the lowFuelLevel event that is automatically converted (if necessary) to the context assumed in the corresponding workflow process definition.

Today some car models are equipped with an emergency call service which is responsible for making an emergency call immediately after airbags are deployed by sending its geographical position. This feature can be improved by including in the call more detailed information that is maintained in the car's portal, such as a damage report (severity), the number of occupants, their medical information, etc.

7.2.4.2 Abstract situations and interaction with external services

Consider the case of a commuter, where she can make better use of her time while *driving to work*. Assuming the following trigger situation where the driver gets into

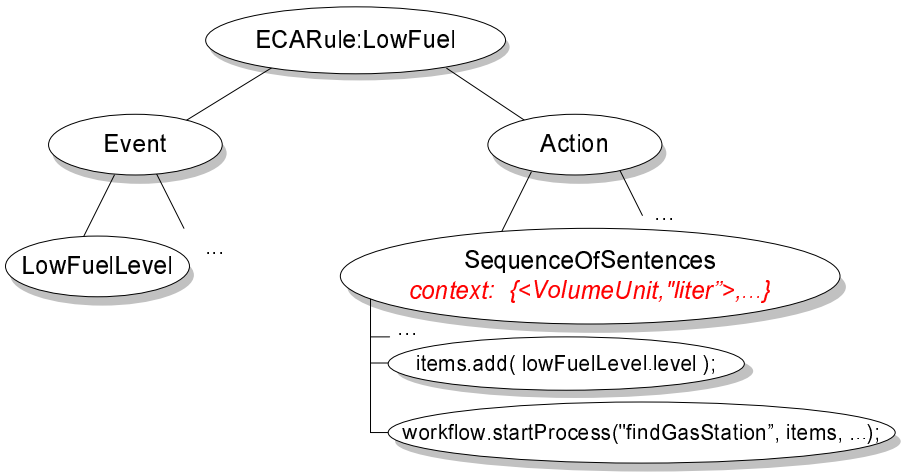

Figure 7.7: Abstract view of the low-fuel rule

the car, it is a workday and the current time is between 8:00am-9:00am. As a reaction the driver is requested to confirm the detected situation and if so the following set of actions can be performed: the best route to work is computed (avoiding traffic jams) and the result is passed to the navigation service; today's scheduled meetings are checked; company news and other personalized news are obtained; and e-mails can be read. Because drivers should concentrate on driving, all this information can be read out by using a text-to-speech service. Of course the set of activities that should be executed while driving to work can be personalized by the driver. Figure 7.8 shows an abstract definition of this rule.

Another useful feature that can be delegated to a rule is the reminder of *changing tires*. In some countries two different kinds of tires are used according to weather demands (winter and summer). Right before the beginning of the season the driver can be reminded and on behalf of the driver an appointment with driver's preferred repair shop can be scheduled, considering the driver's calendar and possibly taking into account the weather forecast.

7.2.4.3 Changes on semantic contexts

As the geographical position of the vehicle is known location-dependent services can be offered. For instance, consider the case of driving a car in Europe. When the car *crosses the border of a country* and taking into account driver's profile –in particular, spoken languages– a bilingual dictionary (and/or a currency convertor) can be loaded in the occupants' PDAs or in the car computer. For this purpose, active capabilities of semantic objects can be explored by subscribing changes on a particular context. To

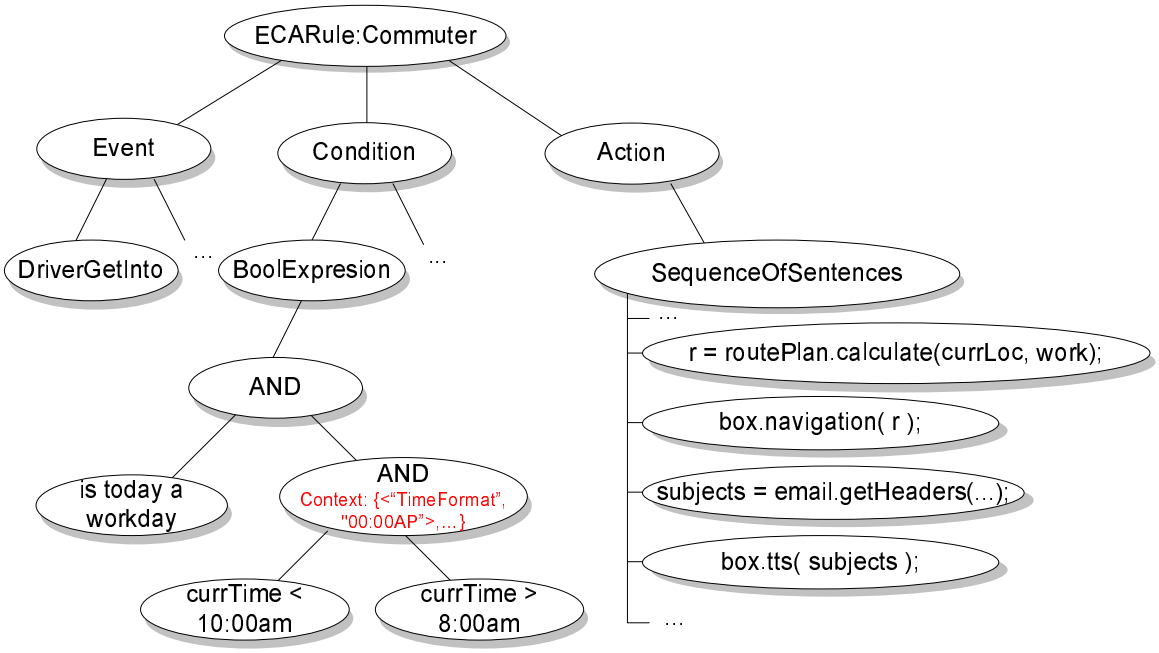

Figure 7.8: Abstract view of the commuter rule

be more specific, the rule subscribes to context changes on the vehicle's geographical position, specifically on the "country"context. Even though the geographical location of the vehicle is updated using GPS coordinates those changes that affect the country context are automatically notified.

Similarly, before the vehicle crosses a country border, a rule can trigger a process that checks the *validation of the documents* of all occupants (e.g. driver license, passport) and also car-related documentation like car insurance or car rental contracts. This can be done, of course, only if relevant information is available on their portals.

7.2.5 Comments & Conclusions

This scenario presents an integral approach to customizing a user's experience in an automobile environment. By means of portals physical entities have a point of web presence. Portal managers, that are responsible for maintaining portal information up-to-date, were enhanced with active capabilities in order to apply user's preferences as a reaction to situations of interest like those described in the previous section. Additionally, it is possible to apply these preferences to diverse vehicles. Heterogeneity problems are solved by using ontologies and meta-data which are an integral part of

the underlying active functionality service. In other words, it was shown how portal mangers of vehicles were endowed with active capabilities.

For this particular scenario two domain-specific ontologies (Vehicle and Profile) were defined where related concepts were specified allowing a clear separation between domain-specific terminology and those terms used by the active functionality infrastructure.

In such a scenario it is also important that rules can be defined according to the characteristics of the users involved (e.g. drivers, manufacturers, car repair shops) where interface alternatives can be provided.

Additionally, rules can contain contextual information which facilitates and simplifies their specification maintaining a high-level rule definition. Because of the integral use of ontologies, every concept (e.g. ECARule, Condition, BoolExpression, SequenceOfSentences) can have a context attached. This context information allows a correct interpretation of data coming from heterogeneous sources and also permits the interaction with external systems by exchanging data in the appropriate form.

Active capabilities of semantic objects were used to react to changes in a context of interest. This empowers the specification of rules by maintaining their definition at a higher level of abstraction.

The publish/subscribe approach used to disseminate events is also an important piece in this scenario. In this way, events of interest are disseminated to all interested consumers by using a concept-based pattern subscription. This communication mechanism facilitates the enhancement of vehicle portal managers providing a loosely-coupled solution by plugging-in a subscriber component. This component is in charge of receiving all event notifications of interest and passing them to the piece of software that is responsible for maintaining portal's state. This characteristic also allows a clear separation between the portal manager and the active functionality.
The active functionality service can run together (in the same computer) with the portal manager, it can run alone in another computer or it can be simply split across several computers in the network. This provides the possibility to best use the computer resources.

Because reactions should take effect immediately in this kind of scenario, two policies were adopted. The first one defines how events should be consumed. The consumption policy was set to *recent* in order to consume the last occurrence of a happening (when more than one of the same kind are available). The second one is related to the moment where the event is consumed and possibly processed. In some situations if the detected situation is too old the reaction may be useless. With this in mind, events have an additional attribute, namely TimeToLive, which defines the time span of validity of events. When this time span expires the event notification is simply discarded.

In the scenario presented in this section, the active functionality service allows defining ECA-rules that react autonomously to specified situations. Because a small set of cues can be detected (typically quantitative variations of dimensions for which they have sensors [Erickson 2002]) in some situations the user should be asked before taking any action. This must be done in order to confirm the detected situation and avoid misunderstandings. For this reason, many of the rules presented here should include in the action part a user confirmation before a rule's action is executed. Under these circumstances, a voice-driven interaction seems to be adequate.

When talking about personalization a discussion about privacy aspects might be relevant but this is out of the scope of this thesis. Here the personalization scenario is presented with the purpose of showing the versatility of the active functionality service.

7.3 Summary

The active functionality service proposed in this thesis was used in two different scenarios with the intention to probe the concepts presented in previous chapters. Its flexibility provides many benefits, namely:

- the definition of domain-specific ontologies separating the active functionality infrastructure from the domain in question,
- different ways of defining rules according to the users involved,
- high-level (context-enabled) definition of rules,
- appropriate interaction with external systems or services by means of plug-ins and with the help of contextual information,
- high-level publish/subscribe communication mechanism thanks to the concept-based approach,
- semantically meaningful data exchange among independent event sources by using sematic objects provided by MIX,
- flexible service deployment by running in diverse environments thanks to the service chain approach.

As mentioned before, the active service presents different "faces"to end-users but the underlying functionality is the same in all cases demonstrating how versatile, flexible and powerful the service is.

Chapter 8

Conclusions and Future Work

Conventional active mechanisms have been designed for centralized systems and are monolithic. This makes it difficult to extend and adapt them to satisfy the requirements of modern applications, e.g. large-scale businesses, Internet-based applications, or emerging pervasive systems. These applications can profit from an active functionality service by encapsulating business semantics into rules enabling quick adaptability to new business requirements and enhancing maintainability. In many cases, only partial database functionality is needed or not required at all. This led to the question of why a full-fledged database system is required when only active functionality and some services of a DBMS are needed. Therefore, the unbundling of active databases was proposed in order to offer an active service that runs decoupled from a database and that may be more suitable for these kinds of applications.

However, unbundling is inadequate for distributed environments since aDBMS components to be "rebundled" were not designed to take into account inherent characteristics of distributed environments like, independent failures, message delays, the lack of a global time, and simultaneity of events. Additionally, the combination of unbundled components and newly developed ones may lead to misinterpretations if the meaning of terms underlying different components is not shared.

The goal of this work was to provide more flexible ECA-rule processing functionality than given by centralized aDBMSs and to support the requirements of other environments, in particular, those of open distributed heterogeneous environments.

To satisfy these requirements the active functionality was conceptually decoupled from the database and a service-based architecture has been proposed in order to offer a flexible and autonomous active service. This proposal is founded on three main pillars: an ontology-based infrastructure, event notifications and service-oriented principles.

Ontologies were used as a common interpretation basis to enable semantically correct interpretation, in this case relying on the MIX model. In this work, ontologies were organized in three layers (basic representation, infrastructure-specific, and domain-specific) with the purpose of clearly separating the infrastructure from the terminology related to the problem that is being solved. The infrastructure-specific ontology reflects the active functionality domain while in the domain-specific ontologies the terminology of particular domains (e.g. Online Auctions, Automotive, etc.) are represented. The ontology approach is not only applied to integrate events from different sources but also to support the interaction among elementary services at semantic level, to empower the addressing of notifications, to represent different timestamp models, and to represent ECA-Rules. Thus, ontologies in this work were used in an integral way.

Another pillar is *event notifications*. An event notification is a message reporting an event to interested consumers. For this purpose, a notification service based on a publish/subscribe paradigm was used, providing asynchronous communications, naturally decoupling producers and consumers, allowing a dynamic number of publishers, and providing location transparency without requiring a name service. Notifications are addressed by using a *concept-based approach* that was introduced in order to provide a higher and common level of abstraction to describe the interests of publishers and subscribers.

As part of the third pillar, the traditional ECA-rule processing was decomposed into elementary services that provide two very simple and generic interfaces where their method arguments are also concepts of the ontology. The rule processing is effectively materialized as a *composition of these elementary services* according to the rule definition. The resulting composition forms a chain of services that are in charge of processing a particular rule. In this way, the flow of work through services can be easily configured – omission or inclusion of services like condition evaluation, event filtering or complex event detection is made easy. The composition of these very simple elementary services provides flexibility by allowing a simple way to configure the flow of processing services that participate in the execution of rules. The interaction among elementary services is accomplished by means of the notification service mentioned above.

The advantages provided for each pillar alone are multiplied when they are combined, offering in this way more benefits than the sum of the parts as described below:

- *Event representation.* Events from different sources are represented using concepts of an ontology and additional contextual information, thus promoting a semantically meaningful data exchange among independent event producers and consumers. Data and events can be integrated thanks to the underlying support of conversion functions which automatically convert data according to the context of its consumer.

- *Event dissemination.* The concept-based publish/subscribe approach used to disseminate events enables the means of communication by maintaining a common and high-level subscription pattern.

- *Rule definition.* Rules can be specified by using different user interfaces (or customized languages) in order to best fit users' characteristics and requirements. The definition of rules can include contextual information enabling a higher level of abstraction that takes care of source-specific event representation peculiarities while allowing the correct interpretation of data involved in events, conditions and actions.

- *Rule representation.* Rules are represented using an intermediate representation which is based on concepts of the ontology (including the use of contextual information). The conceptual rule representation enables the use of a common active mechanism, thus, reusing the same underlying mechanism for different high-level rule definitions. That means, that the active functionality service is logically independent of the end-user's rule definition language.

- *Rule registration.* The registration of new rules does not imply the modification of application code, but the dynamic configuration of elementary services.

- *Service Interaction.* Services interact using an appropriate vocabulary at a semantic level, therefore, avoiding misinterpretations in component interaction.

- *Platform for event composition.* On the basis of a clear separation of concerns, a flexible platform for event composition was proposed to avoid hard-wired event operators within the complex event detector. This approach offers a more flexible alternative where event operators can be plugged into a container which is responsible for controlling the event detection process. Additionally, configurable behavior can be specified with the purpose to respond appropriately in failures, transmission delays, multiple event instances, etc.

- *Deployment of the active service.* The active functionality service (as a whole) can run on different environments ranging from centralized to open distributed environments. This is achieved thanks to the underlying communication mechanism and to the flexible rule processing chain approach. This enables the possibility to best use the available computer resources and the applicability of the active service in a variety of scenarios.

- *External Services.* Rules can interact with diverse external systems or services thanks to the plug-in approach. In this way, new systems can be accessed from rules by developing the corresponding plug-in which can be dynamically plugged into elementary services.

Besides the complete design of the proposed active functionality service a prototype was implemented using the Java Programming Language. This includes elementary services like, a concept-based notification service, an alarm service, a filter service, a condition evaluation service, and an action execution service. Additionally, other software pieces related to the scenarios presented were developed (e.g. Workflow plug-in, e-Mail plug-in, a simple profile manager, the car box, etc).

The Java language was also used to specify ontology concepts and their relationships, thus avoiding any impedance mismatch between programming language and ontology specification language, and allowing the shipping of ontology concepts between different platforms without any further transformations. In addition to data portability, Java supports code portability which is an important issue here since it may be necessary to run rule enforcement in different tiers and on different platforms.

The scenarios we have worked with have clearly shown the flexibility of the active functionality service with the following advantages: the definition of domain-specific ontologies (Online Auctions, Car, Profile) separating the active functionality infrastructure from the domain in question; the versatility to define rules according to the users involved (customized rule definition languages); the ability to attach contextual information in rule definitions; the appropriate interaction with diverse external systems and services (e.g. the car box, e-mail service, workflow engine) thanks to the use of plug-ins; the suitability of the publish/subscribe notification service with concept-based addressing; and of course the semantically meaningful data exchange among independent event producers and consumers.

Because of its conceptual foundation, this architecture promotes extensibility and integration for modern large-scale applications. Flexibility was basically achieved due to the service-oriented architecture where elementary services are composed in order to process the defined rules. Additionally, this architecture encourages the easily adaptation of the active service to satisfy new requirements. In particular, this work provides an extensible platform where the underlying assumptions and the resulting semantics are clearly stated and explicitly defined making its understanding easier. It seems to be an ideal platform, in contrast to one-of-a-kind prototypes, to explore other aspects of active functionality, for instance, interaction with other external services, other implementation features, new event operators, new coupling modes, other timestamp models, just to name a few.

But usually flexibility is achieved at cost of performance. For some scenarios the distributed interaction between elementary services is adequate but for others it may not be. For instance, in real-time centralized scenarios the overhead of using a notification service could be high and consequently the real-time application requirements may not be achieved. On the other side, the underlying notification service based on publish/-

subscribe facilitates the execution of multiple rules in parallel (using various processing units) and in this way performance can be improved.

Among the critical mechanisms for a working service-oriented middleware platform, a notification service, an infrastructure for semantic interoperability and an active functionality service were identified as essential part of modern middleware platforms.

In summary, business rules of modern applications can be defined across applications at a higher and common level of abstraction enhancing extensibility and maintainability, and supporting an effective adaptation to new business requirements.

8.1 Future Work

A dissertation is always limited by the available time. In some cases, therefore, compromises had to be made to obtain pragmatic partial solutions and some interesting questions remained unanswered. These issues will be the subject of future work.

With respect to the event composition approach we can go a step further and try to define event operators at two levels. In the lower one the logic of the event operators and new policies can be defined. At the higher (domain-specific) level all these elements can be reused and combined in order to define new behavior of event operators according to the application in question. All this should be specified by following the integral ontology approach. Additionally, a *testbed* would be necessary to probe the definition of event operators and the composition of composite events. This testbed should support the simulation of the behavior of such event definitions by injecting event instances according to different generation distributions and also by inducing failures with the purpose of analyzing whether the event operator performs as expected.

Another issue of interest is the consideration of events as time intervals (e.g. the execution of methods, activities) and not as points as typically used. This impacts the event algebra that must be adapted to the new model which in turn, results in a new set of operations that can be applied to events, like, overlap, before, during, etc. as proposed in [Allen 1983].

The timestamp ontology must be extended to support the correct interpretation and comparison of timestamps coming from distributed sources. Therefore, different time synchronization dimensions and event observation mechanisms must be studied and properly organized and represented.

New methods to facilitate and support the engineering tasks in the development of event-based systems are needed. The scoping approach presented in [Fiege et al. 2002;

Fiege et al. 2002] seems to be an appropriate direction. From our point of view, a set of basic tools are additionally needed. Visualization tools are useful to analyze the distribution of event generation at different event sources, to graphically trace how event instances flow through their respective rule processing chains, and to have a global overview of the whole running system. This, in fact, could be easily incorporated thanks to the underlying notification service. Thus, the visualization tool can play the role of a consumer that subscribes to events of interest, and this can be realized without requiring any changes in the current implementation.
Tools and techniques for analyzing sets of rules to ensure termination, confluence, and determinism, like those presented in [Aiken et al. 1992; Aiken et al. 1995; Baralis and Widom 2000], are considered as an important area of future research. In particular, the new situation where multiple set of rules are controlled by multiple active mechanisms introduces other challenges that must be investigated.
Because the ECA-Rule Manager determines the rule selection strategy and other aspects related to rule processing, it could be interesting to have the possibility to configure it with different rule processing characteristics.
Other tools related to the management of ontologies are needed. In particular, it could be useful to specify the definition of new concepts by defining them at a higher level and generating the corresponding implementation, i.e. Java code. Moreover, other features like searching and browsing are required. Another interesting issue is try to adapt the notion of conversion functions to the emerging Web-services approach.

Another issue related to the implementation is to replace the notification service which is actually running in the prototype with the one that is being developed in the X^2TS project [Liebig et al. 2000b] that among other things includes transaction support (coupling modes). Because the implementation of the X^2TS is being built for CORBA some changes should be made and additionally the concept-based addressing functionality should be integrated.

It would be interesting to explore other implementation alternatives, for instance, trying to adapt the active functionality service implementation to the J2EE platform. Following the same principles as in the current prototype implementation, elementary services can be implemented as Message-Driven-Beans (MDB), maintaining the interaction among services by means of notifications. In this case, and due to the MDB specification, the subscription pattern of MDBs must be specified at deployment time, restricting the definition of MDBs to subscribe to only one pattern per bean instance. To deal with this problem, configuration (deployment) files can be generated according to the rule in question and consequently new instances of elementary services must be deployed. Another emerging technology of increasing attention is the Web-Service approach which is still not mature enough and undergoing standardization. To offer an implementation of the active functionality service on this platform the communication

protocol specification for asynchronous interactions must be extended to better fulfill asynchronous messaging requirements.

The first impression is that this approach scales better than centralized approaches because of its distributed design which facilitates the addition of (distributed) computation power and avoids bottlenecks and single points of failure. Consequently, when applications need to support increasing amount of notifications and rules, the easiest recourse is to add more computer resources. However, precise performance and scalability experiments are needed to determine how far the underlying messaging technology and elementary services can scale.

Bibliography

Acharya, S., Alonso, R., Franklin, M., and Zdonik, S. 1995. Broadcast Disks: Data Management for Asymetric Communications Environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (1995).

Act-Net Consortium. 1996. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *ACM SIGMOD Record 25*, 3 (Sept.), 40–49. `www.acm.org/sigmod/sigmod-record/9609/adbms.ps`.

Aguilera, M., R.Strom, Struman, D., Astley, M., and Chandra, T. 1999. Matching events in a content-based subscription system. In *Proceedings o fth e18th ACM SIGACT-SIGOPS Symposium on Principles of Distribued Computing* (May 1999), pp. 53–61.

Aiken, A., Hellerstein, J. M., and Widom, J. 1995. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems 20*, 1, 3–41.

Aiken, A., Widom, J., and Hellerstein, J. M. 1992. Behavior of database production rules: termination, confluence, and observable determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (San Diego, California, June 1992), pp. 59–68.

Allen, J. 1983. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM 26*, 11 (Nov.), 832–843.

Bacon, J., Moody, K., and Bates, J. 1998. Opera: Active systems. Technical Report GR/K77068, University of Cambridge - Computer Laboratory. `www.cl.cam.ac.uk/Research/SRG/opera/projects/GRK77068/index.html`.

Baker, D., Cassandra, A., and Rashid, M. 1999. CEDMOS: Complex Event Detection and Monitoring System. Technical Report MCC-CEDMOS-002-99 (March), MCC, Austin, TX. `www.mcc.com/cmi/publications/unclas-techreports/CEDMOS/MCC-CEDMOS-002-99.pdf`.

BANAVAR, G., CHANDRA, T., B.MUKHERJEE, NAGARAJARAO, J., STROM, R., AND STURMAN, D. 1999. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems* (1999), pp. 262–272. computer.org/proceedings/icdcs/0222/02220262abs.htm.

BANKS, A., CHALLENGER, J., CLARKE, P., DAVIS, D., KINGAND, R. P., PARR, F., AND WITTING, K. 2001. Reliable HTTP (HTTPR) Specification, Draft Proposal, Version 1.0. Technical report (July), IBM Research. www-106.ibm.com/developerworks/webservices/library/ws-phtt/httprspecV2.pdf.

BARALIS, E. AND WIDOM, J. 2000. An Algebraic Approach to Static Analysis of Active Database Rules. *ACM Transactions on Database Systems 25*, 3, 269–332.

BATES, J., BACON, J., MOODY, K., AND SPITERI, M. 1998. Using Events for the Scalable Federation of Heterogeneous Components. In *SIGOPS Euroean Workshop on Support for Composing Distributed Applications* (Sintra, Portugal, Sept. 1998). SIGOPS. www.dsg.cs.tcd.ie/~vjcahill/sigops/papers/bates.ps.

BEA SYSTEMS. Application Servers. www.bea.com/products/servers_application.shtml.

BENYOUCEF, M. AND KELLER, R. 2000. An Evaluation of Formalisims for Negotiations in E-Commerce. In *Proc. Workshop on Distributed Communications on the Web*, Volume 1830 of *LNCS* (2000).

BOHACZ, R. On Board Diagnostics (ODB-II). Technical report, High-Tech Performance Magazine. www.dakota-truck.net/OBD2/obd2_high.html.

BORNHÖVD, C. AND BUCHMANN, A. 1999. A Prototype for Metadata-Based Integration of Internet Sources. In *Proceedings Intl. Conference on Advanced Information Systems Engineering (CAiSE)*, Volume 1626 of *LNCS* (Germany, June 1999), pp. 439–445. Springer.

BORNHÖVD, C. AND BUCHMANN, A. 2000. Semantically Meaningful Data Exchange in Loosely Coupled Environments. In *Proc. Intl 6th International Conference on Information Systems Analysis and Synthesis (ISAS'00)* (Orlando, Florida, July 2000).

BORNHÖVD, C., CILIA, M., LIEBIG, C., AND BUCHMANN, A. 2000. An Infrastructure for Meta-Auctions. In *2nd Intl. Workshop on Advance Issues of E-Commerce and Web-based Information Systems (WECWIS'00)* (June 2000), pp. 21–30. IEEE Computer Society.

BORNHÖVD, C. 2000. *Semantic Metadata for the Integration of Heterogeneous Internet Data (in German)*. Ph. D. thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany.

BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., AND MALER, E. 2000. Extensible Markup Language (XML) 1.0 (Second Edition) Specification. Technical report (Oct.), World Wide Web Consortium (W3C). `www.w3.org/TR/REC-xml`.

BUCHMANN, A. P. AND LIEBIG, C. 1999. Distributed, Object-Oriented, Active, Real-Time DBMSs: We Want It All – Do We Need Them (At) All? In *Proceedings of the joint 24th IFAC/IFIP Workshop on Real-Time Programming and 3rd Intl. Workshop on Active and Real-Time Database Systems* (Saarland, Germany, May 1999). Schloss Dagstuhl.

BUCHMANN, A. 1999. *Architecture of Active Database Systems*, Chapter 2, pp. 29–48. Springer. In Paton, N. 1999.

CARREIRO, N. AND GELERNTER, D. 1989. Linda in Context. *Communications of the ACM 32*, 4 (April), 444–458.

CARZANIGA, A., DENG, J., AND WOLF, A. L. 2001. Fast Forwarding for Content-Based Networking. Technical Report CU-CS-922-0 (Nov.), Department of Computer Science, University of Colorado. `www.cs.colorado.edu/users/carzanig/siena/forwarding/index.html`.

CARZANIGA, A., ROSENBLUM, D. R., AND WOLF, A. L. 1999. Challenges for Distributed Event Services: Scalability vs. Expressiveness. In *Engineering Distributed Objects (EDO'99)* (Los Angeles, CA, May 1999). `www.cs.colorado.edu/~carzanig/papers/index.html`.

CARZANIGA, A., ROSENBLUM, D., AND WOLF, A. 2000. Content-based Addressing and Routing: A General Model and its Application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, USA. `www.cs.colorado.edu/~carzanig/papers/cucs-902-00.pdf`.

CARZANIGA, A. 1998. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. Ph. D. thesis, Politecnico di Milano, Milano, Italy. `www.cs.colorado.edu/~carzanig/papers/phd_thesis_a4.ps.gz`.

CASWELL, D. AND DEBATY, P. 2000. Creating Web Representations for Places. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing (HUK2000)* (Bristol, UK, Sept. 2000), pp. 114–126.

CERI, S., GOTTLOB, G., AND TANCA, L. 1990. *Logic Programming and Databases*. Springer.

CERI, S. AND WIDOM, J. 1996. *Applications of Active Databases*, Chapter 10, pp. 259–291. Data Management Systems. Morgan Kaufmann Publishers.

CHAKRAVARTHY, S., LE, R., AND DASARI, R. 1999. ECA Rule Processing in Distributed and Heterogeneous Environments. In Z. TARI, R. MEERSMAN, R. SOLEY, AND O. BUKHERS Eds., *Intl. Symposium on Distributed Objects and Applications (DOA'99)* (Sept. 1999), pp. 330–339.

CHAKRAVARTHY, S. AND MISHRA, D. 1994. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering 14*, 1 (Nov.), 1–26. `ftp.cis.ufl.edu/cis/tech-reports/tr93/tr93-006.ps`.

CHARKRAVARTHY, S., KRISHNAPRASAD, V., ANWAR, E., AND KIM, S. 1994. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (Sept. 1994), pp. 606–617.

CHRISTENSEN, E., CURBERA, F., MEREDITH, G., AND WEERAWARANA, S. 2001. Web Services Description Language (WSDL) 1.1 / Submitted to the World Wide Web Consortium (W3C). Technical report (March), World Wide Web Consortium (W3C). `www.w3.org/TR/wsdl`.

CILIA, M., BORNHÖVD, C., AND BUCHMANN, A. 2001. Moving Active functionality from Centralized to Open Distributed Heterogeneous Environments. In *Proceedings of the 9th IFCIS Conference on Cooperative Information Systems (CoopIS'01)*, Volume 2172 of *LNCS* (Trento, Italy, Sept. 2001), pp. 195–210. Springer.

CILIA, M. AND BUCHMANN, A. 2002. An Active Functionality Service for E-Business Applications. *ACM SIGMOD Record 31*, 1 (March), 24–30. Special Section on Data Management Issues in Electronic Commerce.

CILIA, M., HASSELMEYER, P., AND BUCHMANN, A. 2002. Profiling and Internet Connectivity in Automotive Environments. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'02)* (Hong-Kong, China, Aug. 2002), pp. 1071–1074. Morgan-Kaufmann.

COLLET, C. AND COUPAYE, T. 1996. Composite Events in NAOS. In *Databases and Expert Systems Applications (DEXA)*, Volume 1134 of *LNCS* (Zurich, Switzerland, Sept. 1996), pp. 475–481. Springer.

COLLET, C., VARGAS-SOLAR, G., AND GRAZZIOTIN-RIBEIRO, H. 1998. Towards a Semantic Event Service for Distributed Active Database Applications. In *Databases and Expert Systems Applications (DEXA)*, Volume 1460 of *LNCS* (Sept. 1998), pp. 16–27. Springer. `www.ifi.unizh.ch/staff/vargas/PAPERS/dexa98.ps`.

COLLET, C. 2000. The NODS Project: Networked Open Database Services. In K. D. ET.AL. Ed., *Object and Databases 2000*, Number 1944 in LNCS (2000), pp. 153–169. Springer.

CUBERA, F., DUFTLER, M., KHALAF, R., NAGY, W., MUKHI, N., AND WEERAWARANA, S. 2002. Unraveling the web services web. an introduction to soap, wsdl, and uddi. *IEEE Internet Computing 6*, 2 (March), 86–93. dlib.computer.org/ic/books/ic2002/pdf/w2086.pdf.

CUGOLA, G., NITTO, E. D., AND FUGGETTA, A. 1998. Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)* (1998), pp. 261–270. www.acm.org/pubs/articles/proceedings/soft/302163/p261-cugola/p261-cugola.pdf.

DAYAL, U., BLAUSTEIN, B., BUCHMANN, A., CHAKRAVARTHY, U., HSU, M., LEDIN, R., MCCARTHY, D., ROSENTHAL, A., SARIN, S., CAREY, M., LIVNY, M., AND JAUHARI, R. 1988. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record 17*, 1 (March).

DAYAL, U., BUCHMANN, A., AND MCCARTHY, D. 1988. Rules are Objects Too. In *in Advances in Object-Oriented Database Systems, Proceedings of the 2nd International Workshop on Object-Oriented Database Systems, LNCS 334* (Bad Muenster am Stein, Germany, Sept. 1988), pp. 129–143. Springer-Verlag.

DEMICHIEL, L., YALCINALP, L., AND KRISHNAN, S. 2001. Enterprise JavaBeans. Technical Report Version 2.0 (Aug.), Sun Microsystems, JavaSoftware.

DITTRICH, K., FRITSCHI, H., GATZIU, S., GEPPERT, A., AND VADUVA, A. 2000. SAMOS in Hindsight: Experiences in Building an Active Object-Oriented DBMS. Technical Report 2000.05, Institut fuer Informatik, University of Zurich. ftp.ifi.unizh.ch/pub/techreports/TR-2000/ifi-2000.05.pdf.gz.

ERICKSON, T. 2002. Some Problems with the Notion of Context-Aware Computing. *Communications of the ACM 45*, 2 (Feb.), 102–104.

FABRET, F., LLIRBAT, F., PEREIRA, J., JACOBSEN, A., ROSS, K., AND SHASHA, D. 2001. Filtering Algorithms and Implementation for Very Fast Publish/-Subscribe. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2001), pp. 115–126. www-caravel.inria.fr/LeSubscribe/sigmod01.ps.

FASTOBJECTS. FastObjects j1. www.fastobjects.com/FO_Products_FastObjectsj1_Body.html.

FIEGE, L., MEZINI, M., MÜHL, G., AND BUCHMANN, A. 2002. Engineering Event-Based Systems with Scopes. In B. MAGNUSSON Ed., *In Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*, Volume 2374 of *Lecture Notes in Computer Science* (Málaga, Spain, June 2002), pp. 309–333. Springer-Verlag.

FIEGE, L., MÜHL, G., AND GÄRTNER, F. C. 2002. A Modular Approach to Build Structured Event-based Systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)* (Madrid, Spain, 2002), pp. 385–392. ACM Press.

FIORANO. FioranoMQ. `www.fiorano.com`.

FRITSCHI, H., GATZIU, S., AND DITTRICH, K. 1997. FRAMBOISE - an Approach to Construct Active Database Mechanisms. Technical Report 97.04 (April), Institut fuer Informatik, University of Zurich. `www.ifi.unizh.ch/pub/dbtg/ifi-97.04.ps.gz`.

FRITSCHI, H., GATZIU, S., AND DITTRICH, K. 1998. FRAMBOISE - an Approach to Framework-based Active Data Management System Construction. In *Proceedings of the seventh on Information and Knowledge Management (CIKM 98)* (Maryland, Nov. 1998), pp. 364–370. `www.ifi.unizh.ch/dbtg/Staff/Fritsch/framCIKM98.ps.gz`.

GARCIA-SOLACO, M., SALTOR, F., AND CASTELLANOS, M. 1996. *Semantic heterogeneity in multidatabase systems*, pp. 129–202. Prentice-Hall, Englewood Cliffs, NJ.

GATZIU, S. AND DITTRICH, K. R. 1993. Events in an Active Object-Oriented Database System. In *Proc. 1st Intl. Workshop on Rules in Database Systems (RIDS'93)*, Workshops in Computing (1993), pp. 23–29. Springer.

GATZIU, S., KOSCHEL, A., V. BUETZINGSLOEWEN, G., AND FRITSCHI, H. 1998. Unbundling Active Functionality. *ACM SIGMOD Record 27*, 1 (March), 35–40. `www.cs.umd.edu/areas/db/record/issues/9803/gatziu.ps`.

GATZIU, S. 1994. *Events in an Active Object-Oriented Database System*. Ph. D. thesis, IFI, University of Zurich, Zurich, Switzerland.

GEHANI, N. H., JAGADISH, H. V., AND SHMUELI, O. 1992. Composite event specification in active databases: Model & implementation. In L.-Y. YUAN Ed., *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92)* (Vancouver, Canada, Aug. 1992), pp. 327–338. Morgan Kaufmann.

GEPPERT, A. AND DITTRICH, K. 1998. Bundling: Towards a New Construction Paradigm for Persitence Systems. *Networking and Information Systems Journal, Vol 1(1), June 98 1*, 1 (June), 1–34.

GEPPERT, A. AND TOMBROS, D. 1998. Event-based Distributed Workflow Execution with EVE. In *IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)* (The Lake District, Sept. 1998).

GOH, C. H., BRESSAN, S., MADNICK, S., AND SIEGEL, M. 1999. Context interchange: new features and formalisms for the intelligent integration of information. *ACM Transactions on Information Systems 17*, 3, 270–270.

GRUBER, R., KRISHNAMURTHY, B., AND PANAGOS, E. 1999. The Architecture of the READY Event Notification Service. In *Proceedings of the 19th IEEE Intl. Conf. on Distributed Computing Systems Middleware Workshop* (Austin, Texas, May 1999). IEEE. `www.research.att.com/~ready`.

GRUBER, T. R. 1995. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. *Int. Journal of Human-Computer Studies (IJHCS) 43*, 5/6, 907–928.

GUARINO, N. 1997. Understanding, Building and using Ontologies. *Int. Journal of Human-Computer Studies (IJHCS) 46*, 2/3 (Feb.), 293–310.

GUDGIN, M., HADLEY, M., MOREAU, J.-J., AND NIELSEN, H. F. 2001. Simple Object Access Protocol (SOAP) 1.2 / Submitted to the World Wide Web Consortium (W3C). Technical report (July), World Wide Web Consortium (W3C). `www.w3.org/TR/SOAP`.

HADA, S. AND MARUYAMA, H. 2000. SOAP Security Extensions. Technical report (Nov.), Tokyo Research Laboratory, IBM Research. `www.trl.ibm.com/projects/xml/soap/wp/wp.html`.

HAKIMPOUR, F. AND GEPPERT, A. 2001. Resolving Semantic Heterogeneity in Schema Integration: an Ontology Based Approach. In *Proceedings of the Intl Conference on Formal Ontology in Information Systems (FOIS'01)* (Ogunquit, Maine, Oct. 2001), pp. 297–308. ACM Press.

HAMMER, M. AND SARIN, S. K. 1978. Efficient Monitoring of Database Assertions. In E. I. LOWENTHAL AND N. B. DALE Eds., *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (Austin, Texas, May 1978), pp. 159. ACM Press.

HANSMANN, U., MERK, L., NICKLOUS, M., AND STOBER, T. 2001. *Pervasive Computing Handbook*. Springer. ISBN 3-540-67122-6.

HAPNER, M., BURRIDGE, R., AND SHARMA, R. 1999. Java Message Service. Specification Version 1.0.2 (Nov.), Sun Microsystems, JavaSoftware.

Hewlett-Packard. 2001. Web Services Concepts – A Technical Overview. `www.e-speak.hp.com/media/techoverview.pdf`.

HP BLUESTONE. 2001. Core Service Framework (CSF). www.bluestone.com/PRODUCTS/core_services_framework/.

HP BLUESTONE. Application servers. www.hp.com/go/webservices/.

IBM CORP. Application servers. www-4.ibm.com/software/webservers/.

IBM. MQ-Series. www-4.ibm.com/software/ts/mqseries/.

ISO. 1989. Road vehicles – Diagnostic systems – Requirements for interchange of digital information. Technical report, International Standards Organization (ISO).

KIM, W., CHOI, I., GALA, S. K., AND SCHEEVEL, M. 1993. On Resolving Schematic Heterogeneity in Multidatabase Systems. *Distributed and Parallel Databases 1*, 3 (July), 251–279.

KIM, W. AND SEO, J. 1991. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer Magazine 24*, 12 (Dec.), 12–18.

KINDBERG, T., BARTON, J., MORGAN, J., BECKER, G., CASWELL, D., P.DEBATY, GOPAL, G., FRID, M., KRISHNAN, V., MORRIS, H., SCHETTINO, J., SERRA, B., AND SPASOJEVIC, M. 2000. People, Places, Things: Web Presence for the Real World. In *Proceedings IEEE Workshop on Mobile Computing Systems and Application (WMCSA 2000)* (Monterey, CA, Dec. 2000). IEEE Computer Society Press.

KOPETZ, H. 1992. Sparse Time versus Dense Time in Distributed Real-Time Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS)* (Yakohama, Japan, June 1992), pp. 460–467.

KOSCHEL, A., GATZIU, S., VON BÜLTZINGSLOEWEN, G., AND FRITSCHI, H. 1999. *Unbundling active functionality*, Chapter 10. IDEA Group Publishing.

KOSCHEL, A., KRAMER, R., V. BUELTZINGSLOEWEN, G., BLEIBEL, T., KRUMLINDE, P., SCHMUCK, S., AND WEINAND, C. 1997. Configurable Active Functionality for CORBA. In *CORBA: Implementation, Use and Evaluation. ECOOP'97 Workshop* (Finland, June 1997). www.fzi.de/dbs/publications/Koschel/KoKrBuBlKrScWe97-EcoopWS.ps.gz.

KOSCHEL, A. AND LOCKEMANN, P. 1998. Distributed Events in Active Database Systems - Letting the Genie out of the Bottle. *Data & Knowledge Engineering 25*, 1-2 (March), 29–53.

KOTTIG, A. 2000. Realisierung einier generischen Wrapper-Komponente zur Abbildung von XML-Daten auf ein ontologiebasiertes Datenmodell (in German). Master's thesis, Darmstadt University of Technology - Computer Science Department, Darmstadt, Germany.

LAMPORT, L. 1980. The "Hoare Logic" of Concurrent Programs. *Acta Informatica 14*, 21–37.

LIAO, H. 1997. Global Events in Sentinel: Design and Implementation of a Global Event Detector. Master's thesis, CISE, University of Florida, Gainesville, Florida.

LIEBIG, C., CILIA, M., BETZ, M., AND BUCHMANN, A. 2000. A publish-subscribe corba persistent state service prototype. In *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms and Open Distribued Processing (Middleware 2000)*, Volume 1795 of *LNCS* (New York, USA, April 2000), pp. 231–255. Springer.

LIEBIG, C., CILIA, M., AND BUCHMANN, A. 1999. Event Composition in Time-dependent Distributed Systems. In *Proceedings 4th IFCIS Conference on Cooperative Information Systems (CoopIS'99)* (Edinburgh, Scotland, Sept. 1999), pp. 70–78. IEEE Computer Society Press.

LIEBIG, C., MALVA, M., AND BUCHMANN, A. 2000a. Integrating Notifications and Transactions: Concepts and X2TS Prototype. In *Proceedings Intl. Workshop on Engineering Distributed Objects (EDO)*, Volume 1999 of *LNCS* (Nov. 2000), pp. 194–214. Springer.

LIEBIG, C., MALVA, M., AND BUCHMANN, A. 2000b. X^2TS: Unbundling Active Object Systems (Short Paper). In *IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'00)*, Volume 1795 of *LNCS* (April 2000). Springer.

LIEBIG, C. AND TAI, S. 2001. Middleware Mediated Transactions. In *Proceedings 3rd Intl. Symposium on Distributed Objects and Applications (DOA'00)* (Sept. 2001). IEEE Computer Society Press.

LOPES DE OLIVEIRA, J., MEDEIROS, C. B., AND CILIA, M. 1997. Active Customization of GIS User Interfaces. In *Proceedings of the Thirteenth International Conference on Data Engineering (ICDE)* (Birmingham, U.K., April 1997), pp. 487–496. IEEE Computer Society Press.

MA, C. AND BACON, J. 1998. COBEA: A CORBA-based Event Architecture. In *Conference on Object-Oriented Technologies and Systems (COOTS'98)* (New Mexico, USA, April 1998), pp. 117–131. USENIX.

MILLS, D. L. 1990. On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet System. *ACM Computer Communication Review 20*, 1.

MILLS, D. L. 1992. Network Time Protocol - Version 3. Technical Report RFC-1305 (March), Network Working Group, University of Delaware.

MÜHL, G., FIEGE, L., AND BUCHMANN, A. 2002. Filter Similarities in Content-Based Publish/Subscribe Systems. In H. SCHMECK, T. UNGERER, AND L. WOLF Eds., *International Conference on Architecture of Computing Systems (ARCS)*, Volume 2299 of *Lecture Notes in Computer Science* (2002), pp. 224–238. Springer-Verlag.

MÜHL, G. 2001. Generic Constrains for Content-based Publish/Subscribe. In *Proceedings of the 6th Intl Conference on Cooperative Information Systems (CoopIS)*, Volume 2172 of *LNCS* (Trento, Italy, Sept. 2001), pp. 211–225. Springer.

OBJECT MANAGEMENT GROUP. 1997. Event Service Specification. Technical Report formal/97-12-11 (May), Object Management Group (OMG), Famingham, MA.

OBJECT MANAGEMENT GROUP. 1998. CORBA Notification Service Specification. Technical Report telecom/98-06-15 (May), Object Management Group (OMG), Famingham, MA.

OBJECT MANAGEMENT GROUP. 2001. Internet Inter-ORB Protocol (IIOP) Specification. `www.omg.org/technology/documents/formal/corba_iiop.htm`.

OKI, B., PFLUEGL, M., SIEGEL, A., AND SKEEN, D. 1993. The Information Bus – An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th Symposium on Operating Systems Principles (SIGOPS)* (USA, Dec. 1993), pp. 58–68.

OPYRCHAL, L., ASTLEY, M., AUERBACH, J., BANAVAR, G., STROM, R., AND STURMAN, D. 2000. Exploiting IP Multicast in Content-based Publish-Subscribe Systems. In J. SVENTEK AND G. COULSON Eds., *Middleware 2000*, Volume 1795 of *LNCS* (2000), pp. 185–207. Springer.

OUKSEL, A. AND SHETH, A. 1999. Semantic Interoperability in Global Information Systems - A Brief Introduction to the Research Area and the Special Section. *ACM SIGMOD Record 28*, 1 (March), 5–12. `www.acm.org/sigmod/record/issues/9903/special/intro.pdf.gz`.

PATON, N. Ed. 1999. *Active Rules in Database Systems*. Springer.

PILIOURA, T. AND TSALGATIDOU, A. 2001. E-Services: Current Technology and Open Issues. In *Proceedings of the International Workshop on Technologies for E-Services (TES)*, Volume 2193 of *LNCS* (Rome, Italy, Sept. 2001), pp. 1–15. Springer.

SCHWIDERSKI, S. 1996. *Monitoring the Behaviour of Distributed Systems*. Ph. D. thesis, Selwyn College, Computer Lab, University of Cambridge, Computer Labs, United Kingdom.

SEGALL, B. AND ARNOLD, D. 1997. Elvin has Left the Building: A Publish/-Subscribe Notification Service with Quenching. In *Australian Unix Users Group Annual Conferece (AUUG'97)* (July 1997).

SOCIETY OF AUTOMOTIVE ENGINEERS. 1994. SAE Vehicle Network for Multiplexing and Data Communications. Technical report, Standards Committee, SAE J1850 Standard. `www.sae.org`.

SONICSOFTWARE. SonicMQ. `www.sonicsoftware.com/products/`.

SPIRITSOFT. Spirit Lite. `www.spirit-soft.com/products/lite/overview.html`.

SUN MICROSYSTEMS. 2001. Java 2 Enterprise Edition Platform Specification. Technical Report Version 1.3 (Aug.), Sun Microsystems, JavaSoftware.

TALARIAN. SmartSockets for JMS. `www.talarian.com/products/jms/index.shtml`.

TIBCO. TIB/Rendezvous. `www.rv.tibco.com`.

TIME SERVICE DEPARTMENT. U.S. Naval Observatory. `tycho.usno.navy.mil`.

UDDI. Universal Description, Discovery and Integration (UDDI). `www.uddi.org`.

VARGAS-SOLAR, G. 2000. Flexible Event Service for Integrating Distributed Database Applications. In *Proceedings of the EDBT PhD Workshop* (Konstanz, Germany, March 2000). `www.ifi.unizh.ch/staff/vargas/PAPERS/wedbt.ps`.

WIDOM, J. AND CERI, S. Eds. 1996. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Data Management Systems. Morgan Kaufmann Publishers.

YANG, S. AND CHAKRAVARTHY, S. 1999. Formal Semantics of Composite Events for Distributed Environments. In *proceedings of the 15th International Conference on Data Engineering (ICDE'99)* (Sydney, Australia, March 1999), pp. 400–407. IEEE Computer Society Press.

ZIMMERMANN, J. AND BUCHMANN, A. 1999. *REACH*, Chapter 14, pp. 263–277. Springer. In Paton, N. 1999.

ZIMMER, D. AND UNLAND, R. 1999. On the Semantics of Complex Events in Active Database Management Systems. In *proceedings of the 15th International Conference on Data Engineering (ICDE'99)* (Sydney, Australia, March 1999), pp. 392–399. IEEE Computer Society Press.

Appendix A

Background

As this work involves a wide variety of topics, this appendix intends to present the essentials related to them. Below follows a description of ECA-rule processing, publish/subscribe messaging and e-services.

A.1 Processing ECA-Rules

Rule execution semantics prescribe how an active system behaves once a set of rules has been defined. Rule execution behavior can be quite complex, but we restrict ourselves to describing only essential aspects here. For a more detailed description see [Widom and Ceri 1996; Act-Net Consortium 1996].

All begins with event instances signaled by event sources that feed the complex event detector, which selects and consumes these events. *Consumption modes* [Charkravarthy et al. 1994] determine which of these event instances are considered for firing rules. The two most common modes are recent and chronicle. In the former, the most recent event occurrences are used, while in the latter, the oldest event occurrences are consumed. Notice that in both cases a temporal order of event occurrences is required. Different consumption modes may be required by different application classes.

Usually there are specific points in time at which rules may be processed during the execution of an active system. The *rule processing granularity* specifies how often these points occur. For example, the finest granularity is "always" which means that rules are processed as soon as any rule's triggering event occurs. If we consider the database context, rules may be processed after the occurrence of database operations (small), data manipulation statements (medium), or transactions (coarse).

At granularity cycles and only if rules were triggered, the *rule processing algorithm* is invoked. If more than one rule was triggered, it may be necessary to select one after the other from this set. This process of *rule selection* is known as *conflict resolution*, where basically three strategies can be applied: a) one rule is selected from the fireable pool, after rule execution the set is determined again, b) sequential execution of all rules in an evaluation cycle, and c) parallel execution of all rules in an evaluation cycle.

After rules are selected, their corresponding conditions are evaluated. Notice that conditions can be expressed as predicates on database states using a query language, and also external method invocations can be used. The specification of conditions can involve variables that will be bound at runtime with the content of triggering events. If a condition evaluates to true, then the action associated with this rule must be executed. Actions can be any sequence of operations on or outside of a database. These operations can include attributes of the triggering event.

For some applications it may be useful to delay the evaluation of a triggered rule's condition or the execution of its action until the end of the transaction, or it may be useful to evaluate a triggered rule's condition or execute its action in a separate transaction. These possibilities yield to the notion of *coupling modes*. One coupling mode can specify the transactional relationship between a rule's triggering event and the evaluation of its condition while another coupling mode can specify the transactional relationship between a rule's condition evaluation and the execution of its action. The originally proposed coupling modes are immediate, deferred and decoupled [Dayal et al. 1988].

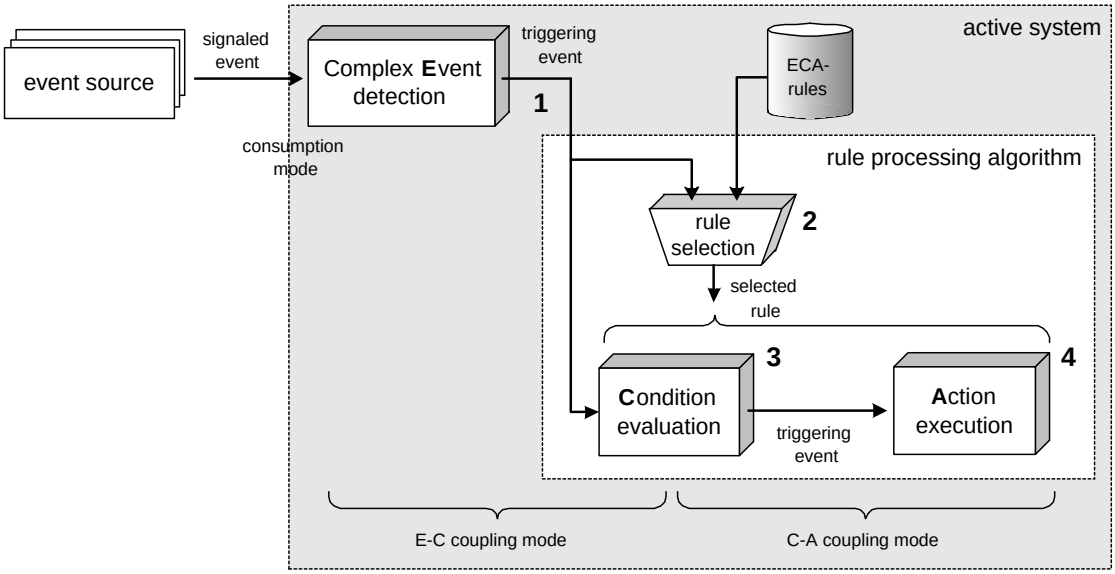

Figure A.1: Schematic view of the ECA-Rule processing mechanism

To sum up, an ECA-rule processing mechanism can be formulated as a sequence of four steps (as illustrated in Figure A.1):

1. *Complex event detection*: it selects instances of signaled events in order to detect specified situations of interest where various event occurrences may be involved. As a result, these picked instances are bound together and signaled as a (single) complex event.

2. *Rule selection*: according to the triggering events, fireable rules are selected. If there are multiple a conflict resolution policy must be applied.

3. *Condition evaluation*: selected rules receive the triggering event as a parameter, thus, allowing the condition evaluation code to access event content. Transaction dependencies between event detection and the evaluation of a rule's condition are specified using Event-Condition coupling modes.

4. *Action execution*: if the rule's condition evaluates to true, the corresponding action is executed taking the triggering event as a parameter. Transaction dependencies are specified similarly to step 3.

It should be noted that step 1 (complex event detection) can be skipped if rules involve primary events only.

A.2 Publish/Subscribe Messaging

The publish/subscribe model is a very general communication model in distributed computing. Its main features can be characterized by natural multicast functionality and decoupling of producers and consumers. These features are explained below.

A.2.1 Natural Multicast Functionality

The publish/subscribe model supports distributed computing where one application can send the same message once and multiple applications receive it. This model contains basically two main players: information *producers*, which publish data to the system (or network), and information *consumers*, which subscribe to particular categories of data within the system. Producers are responsible for initiating the delivery of information in the form of messages (or notifications). The system ensures the timely

delivery of published events to all interested consumers. These consumers get information by using event callback functions that are triggered when messages arrive. Notice that applications can play the role of a producer and a consumer of messages at the same time.

A.2.2 Decoupling of Producers and Consumers

In addition to supporting many-to-many communication, the primary requirement met by publish/subscribe systems is that producers and consumers of messages are anonymous to each other, so the number of publishers and subscribers may dynamically change, and individual publishers and subscribers may evolve without disrupting the existing system.

Publish/subscribe allows a natural decoupling of producers and consumers of messages. Instead of addressing by physical location (i.e. network address, socket number, server identity), a publish/subscribe interaction offers three generic alternatives in order to address messages: channels, subjects and content.

The first option was adopted by the first generation of publish/subscribe systems [Object Management Group 1997]. They used a *channel* to communicate producers with consumers. Messages were published to a specific channel by producers and delivered to all consumers that had subscribed to it. Channels allow an efficient data delivery but subscription expressiveness is limited. Figure A.2 shows an example where sport news are delivered from producers to consumers by means of a channel.

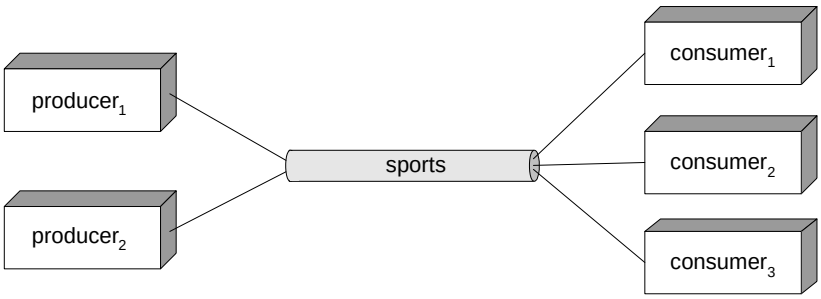

Figure A.2: Channel-based Addressing

In the second alternative a *subject* is associated to each message [Oki et al. 1993][TIBCO]. Subject names consist of one or more elements (usually a string) organized in a tree by means of a dot notation. Subject-based addressing features a set of rules that defines a uniform name space for messages and their destinations. This approach is inflexible

if changes to the subject organization are required, implying fixes in all participant applications.

In Figure A.3 two producers are shown when publishing messages. Notice that associated to each message are their corresponding subjects symbolized here using tags/labels. Subscriptions are carried out using the subject name space organization where wildcards can be used to specify consumer interests. For instance, $consumer_1$ subscribes to all sports news (by means of news.sports.*), $consumer_2$ to all kind of news (news.>), where $consumer_3$ subscribes specifically to the stock prices of SUNW (news.finance.-stocks.SUNW). After producers publish their messages (as depicted in the figure), $consumer_1$ receives one notification with the result of a basketball game, $consumer_3$ receives a notification related to the stock price of SUNW and $consumer_2$ receives both notifications.

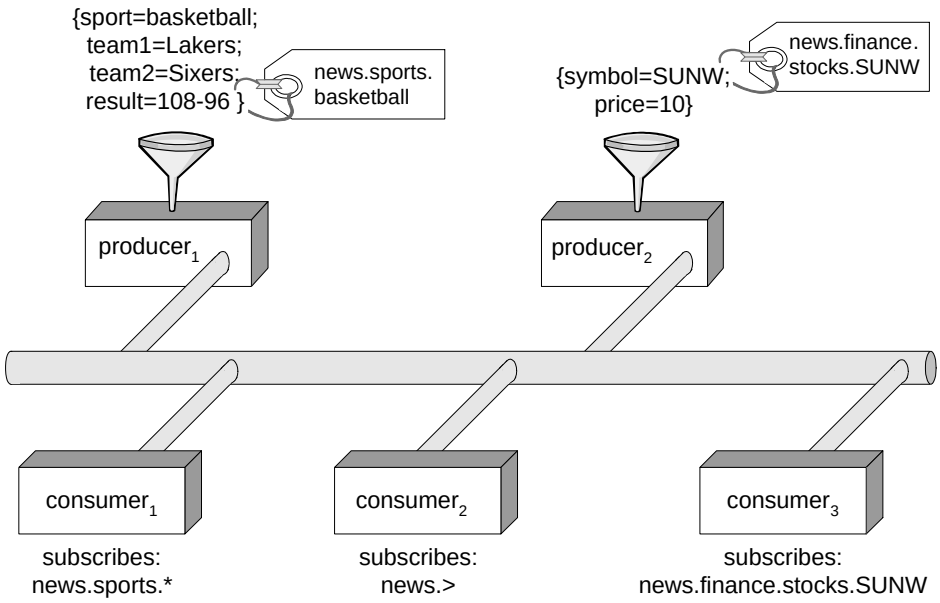

Figure A.3: Subject-based Addressing

Finally, the third option is the *content-based* approach which allows the use of the whole content of a message for filtering [Carreiro and Gelernter 1989; Object Management Group 1998; Aguilera et al. 1999; Mühl 2001]. This alternative is a more flexible mechanism but requires a complex infrastructure.

In all the options mentioned above, producers do not have any knowledge of who or what applications are subscribing. Additionally, the physical location of message consumers becomes entirely transparent without requiring a name service (location transparency).

Message Oriented Middleware (MOM) products and standard specifications like CORBA Notification Service [Object Management Group 1998] and Java Message Service (JMS) [Hapner et al. 1999] support the publish/subscribe model.

A.3 e-Services

E-Services are self-contained, modular applications that can be described, published, located and invoked over a network [Pilioura and Tsalgatidou 2001]. E-Services can be seen as the evolution from object-oriented systems to systems of services. As in object-oriented systems, some of the fundamental concepts in e-services are encapsulation, message passing and dynamic binding. Even though e-Services are not a radically new development paradigm, they are, in many aspects, simply an extension of the current trend toward component-based distributed computing where components are (large and) loosely coupled.

The e-services conceptual model is comprised of three participants and three fundamental operations (as depicted in Figure A.4). The participants are the following:

- A *service provider* is a network node that provides a service interface for a software asset that manages a specific set of tasks. A service provider node can represent the services of a business entity or it can simply represent the service interface for a reusable subsystem.
- A *service requestor* is a network node that discovers and invokes other software services to provide a (business) solution. Service requestor nodes will often represent a business application component that performs remote procedure calls to a distributed object, the service provider.
- The *service broker* is a network node that acts as a repository, e.g. yellow pages, for software interfaces that are published by service providers.

These three participants interact using three basic operations: publish, find and bind. Service providers *publish* services to a service broker. Service requesters *find* required services using a service broker and if they have found a suitable service, they *bind* to them.

Services require a platform that needs to support two key requirements for a service-based infrastructure technology: standard-based service interactions and isolation of service implementations. As usual in many runtime environments, it is expected that the platform handles transactions, security, availability and scalability, but it should

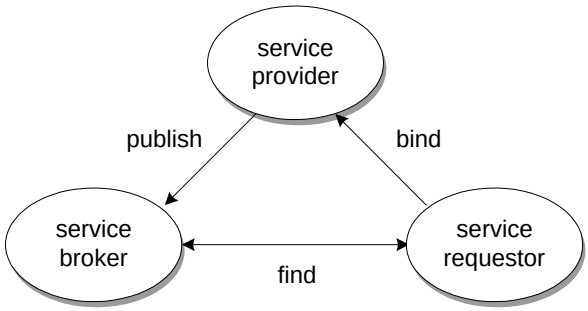

Figure A.4: e-Service Conceptual Model

provide additional capabilities for registration, location, interaction and monitoring of services. A representative of an e-Service platform is the HP Core Service Framework (CSF) [HP Bluestone 2001] which is explained in more detail in Section 6.2.1.

Web-services can be seen as a natural evolution of e-Services. They are essentially Internet-oriented, modular and reusable components that are created by wrapping a business application inside a Web service interface [Hewlett-Packard 2001]. Web-services comprise a set of platform-neutral technologies designed to ease the delivery of network services over intranets and the Internet. Cross-platform capabilities are one of the Web-Services key attractions. They enable interoperability via a set of open standards which distinguish them from previous network services such as CORBA's Internet Inter-ORB Protocol (IIOP) [Object Management Group 2001]. Those open Internet standards are briefly briefly described below:

- Extensible Markup Language (XML) [Bray et al. 2000]: XML is a text-based markup language which uses tags for describing presentation and data. XML is used for the definition of portable structured data. It can be used as a language for defining data interchange formats, such as markup grammars or vocabularies and interchange formats and messaging.

- Simple Object Access Protocol (SOAP) [Gudgin et al. 2001]: It is an XML-based lightweight protocol for the exchange of information in a decentralized, distributed environment. SOAP defines a messaging protocol between requestor and provider objects. SOAP is platform, operating system, object model and programming language independent. However, the transport and language bindings as well as data-encoding are all implementation dependent. Additionally, some work is being done in order to provide reliability and security to this protocol (reliable HTTP [Banks et al. 2001] and SOAP security extensions [Hada and Maruyama 2000]).

- Web Services Description Language (WSDL) [Christensen et al. 2001]: This language is an XML-based language that provides a standard way of describing service interfaces (e.g. IDLs). It provides a simple way for service providers to describe the format of requests and response messages for remote method invocation (RMI). WSDL addresses the topic of service interface in a way that is independent of the underlying protocol and encoding requirements, thus, providing an abstract language for defining the published operations of a service with their respective parameters and data types. The language also addresses the binding details of the service.

- Universal Description, Discovery and Integration (UDDI) [UDDI]: This specification was outlined to help facilitate the creation, description, discovery and integration of Web-based services. UDDI takes an approach that relies upon a distributed registry of services descriptions (implemented in a common XML format, e.g. WSDL). It defines common means to publish information about business and services. It can be used to locate not only technical details about how to interact with a particular service, but also information at a business level.

In [Cubera et al. 2002] a clear overview of these standards and how they play together is presented.

Application servers play an important role as a Web-Service platform. This is founded on their ability to assemble different types of back-end applications, and because they also offer a high-end run-time infrastructure to manage transactions, scalability and security issues. For this reason, major players in the application server arena are adapting their products to satisfy new requirements [IBM Corp.; HP Bluestone; BEA Systems] and to comply with (evolving) open standards.

Appendix B

Ontology Definition - Infrastructure

B.1 Basic Representation (Represent)

SimpleSemanticObject is-a Represent.SemanticObject
DESCRIPTION: Represents a container class for simple semantic objects, i.e. semantic objects that only have one attribute value.

ComplexSemanticObject is-a Represent.SemanticObject
DESCRIPTION: Represents a container class for complex semantic objects, i.e., semantic objects that have multiple attribute values. Each of these attributes is itself a semantic object and is identified by it's associated concept name assuming that every attribute in a semantic object can be mapped to a unique corresponding concept.

Bool is-a Represent.SimpleSemanticObject
DESCRIPTION: Provides an object wrapper to represent a value of the primitive type boolean.

CharString is-a Represent.SimpleSemanticObject
DESCRIPTION: Represents a a growable buffer for characters.

NumberObject is-a Represent.SimpleSemanticObject
DESCRIPTION: Is an abstract representation concept for numeric scalar types. *IntegerNumber*, *LongNumber*, *FloatNumber* and *DoubleNumber* are specializations of *NumberObject* that bind to a particular numeric representation.

FloatNumber is-a Represent.NumberObject
DESCRIPTION: Provides an object wrapper to represent float data values, and serves as a place for float-oriented operations.

IntegerNumber is-a Represent.NumberObject
DESCRIPTION: Provides an object wrapper for integer values (in decimal notation).

LongNumber is-a Represent.NumberObject
DESCRIPTION: Provides an object wrapper for long data values and serves as a place for long-oriented operations.

URL is-a Represent.CharString
DESCRIPTION: Represents a Uniform Resource Locator, a pointer to a "resource"on the World Wide Web.

Currency is-a Represent.CharString
DESCRIPTION: Represents a constant that serves as a unit of measurement for dimension monetary.

CurrencyCode is-a Represent.CharString
DESCRIPTION: Provides the type of a currency coding according to the ISO 4217. For instance, the ThreeLetterCurrencyCode USD corresponds to U.S. Dollar.
DOMAIN: {"ThreeLetterCurrencyCode", "FullCurrencyName"}.

DateTime is-a Represent.CharString
DESCRIPTION: Provides a concept for the representation of a point in time.

DateTimeFormat is-a Represent.CharString
DESCRIPTION: Specifies the representation format of a *DateTime* object, e.g., "DD.MM.YYYY, HH:MM:SS".

TimeZone is-a Represent.CharString
DESCRIPTION: Specifies the respective time zone in regard to which a given *DateTime* object has to be interpreted.

PhysicalQuantity is-a Represent.FloatNumber
DESCRIPTION: Represents a measure of some quantifiable aspect of the modeled world, such as the distance of two cities is a PhysicalQuantity (of dimension length). A PhysicalQuantity is distinguished from a purely numeric entity like a real number by having a *PhysicalDimension*, a *UnitOfMeasure*, and a *Scale* associated with it.

PhysicalDimension is-a Represent.CharString
DESCRIPTION: Represents a property that is associated with a *PhysicalQuantity* for purposes of classification or differentiation. Mass, Length, Time, Energy, and Force are examples of physical dimensions.

UnitOfMeasure is-a Represent.CharString
DESCRIPTION: Specifies a constant quantity that serves as a standard of measurement for some dimension. For example, the meter and inch are units of measure for the length dimension.

SystemOfUnits is-a Represent.SetOfConstants
DESCRIPTION: Is a set of *UnitsOfMeasure*s that defines a standard system of measurement, e.g. metric, imperial. Each instance of the set is a canonical *UnitOfMeasure* for a *PhysicalQuantity*.

Scale is-a Represent.FloatNumber
DESCRIPTION: Specifies the scale factor of a given numerical value.

Price is-a Represent.PhysicalQuantity
DESCRIPTION: Is physical quantity of dimension monetary that represents a given amount of value.

Distance is-a Represent.PhysicalQuantity
DESCRIPTION: Is physical quantity of dimension length.

TimeAmount is-a Represent.PhysicalQuantity
DESCRIPTION: Specifies a given amount of time.

SetOf is-a Represent.SimpleSemanticObject
DESCRIPTION: Provides an object wrapper to represent a set of semantic objects.

SequenceOf is-a Represent.SimpleSemanticObject
DESCRIPTION: Provides an object wrapper to represent a (double linked) list of semantic objects.

...

B.2 Infrastructure-specific Ontology (Infra)

ECARule is-a Represent.ComplexSemanticObject
DESCRIPTION: Identifies an Event-Condition-Action rule meaning: when the specified *Event* occurs, check the *Condition* and if it holds, execute the *Action*. The use of a *Condition* is optional.
REQUIRED ATTRS.: Infra.Event, Infra.Action, Infra.RuleId

RuleId is-a Represent.CharString
DESCRIPTION: Represents an *ECARule* identification.

RuleName is-a Represent.CharString
DESCRIPTION: Represents the name of a particular rule.

RuleDescription is-a Represent.CharString
DESCRIPTION: Represents a textual description of a particular rule.

Event is-a Represent.ComplexSemanticObject
DESCRIPTION: Represents a happening of interest.

PrimitiveEvent is-a Infra.Event
DESCRIPTION: Represents an elementary occurrence of interest.

TemporalEvent is-a Infra.PrimitiveEvent
DESCRIPTION: Represents a time-related happening.

AbsoluteTemporalEvent is-a Infra.TemporalEvent
DESCRIPTION: Identifies an absolute point in time, e.g. 16/March/2001 14:00.
REQUIRED ATTRS.: Represent.DateTime

PeriodicEvent is-a Infra.TemporalEvent
DESCRIPTION: Identifies a temporal event that occurs regularly, e.g. every Friday at 19:30.
REQUIRED ATTRS.: Infra.BeginOfPeriod, Infra.Periodicity

RelativeEvent is-a Infra.TemporalEvent
DESCRIPTION: Identifies a happening which is relative to a reference *Event*, e.g. one week after *StartOfAuction*.
REQUIRED ATTRS.: Infra.EventOfReference, Represent.TimeAmount

CompositeEvent is-a Infra.Event
DESCRIPTION: Identifies a combination of other primitive or composite events using a set of event operators.
REQUIRED ATTRS.: Infra.EventOperator, Infra.SetOfEvents, Infra.ConsumptionMode

EventOperator is-a Represent.ComplexSemanticObject
DESCRIPTION: Represents the operator that compose events conforming a composite event. Usually, this composition relies on an event algebra that may include operators for sequence, disjunction, conjunction, negation, etc.
REQUIRED ATTRS.: Infra.OperatorLogic, Infra.UncertaintyPolicy, Infra.SelectionPolicy, Infra.FailureHandlingPolicy

ConsumptionMode is-a Represent.CharString
DESCRIPTION: Determines which event instance (from a queue of event instances) is considered for consumption. This is normally determined using the timestamp of involved events.
DOMAIN: {"chronicle", "recent"}

Condition is-a Represent.ComplexSemanticObject
DESCRIPTION: Represents a boolean predicate with the purpose of ensuring some particular aspects once a rule is fired and before its action is executed. It can involve notifications' attributes and external (boolean) functions.
REQUIRED ATTRS.: Infra.BooleanExpression, Infra.CouplingMode

Filter is-a Represent.ComplexSemanticObject
Description: Represents a boolean predicate that is restricted to involve attributes of notifications. Normally used to reduce network traffic by discarding notifications that are not of interest (do not hold the predicate).
Required Attrs.: Infra.BooleanExpression

Action is-a Represent.ComplexSemanticObject
Description: Represents a sequence of statements that are executed as part of the action of an *ECARule*.
Required Attrs.: Infra.SequenceOfStatements, Infra.CouplingMode

CouplingMode is-a Represent.CharString
Description: Specifies the transactional relationship between a rule's triggering event and the evaluation of its condition or between a rule's condition evaluation and the execution of its action.
Domain: {"immediate", "decoupled", "deferred", "none"}

Timestamp is-a Represent.ComplexSemanticObject
Description: Identifies a point in time. It is normally used to capture the moment at which a happening occurs.

Notification is-a Represent.ComplexSemanticObject
Description: Represents a message reporting an occurrence of an event to interested consumers.
Required Attrs.: Infra.Event, Infra.OperationalData

OperationData is-a Represent.ComplexSemanticObject
Description: Identifies those information related to the notification of an *Event*, like event source, occurrence time, etc.
Required Attrs.: Infra.EventSource, Infra.OccurrenceTime

OccurrenceTime is-a Represent.ComplexSemanticObject
Description: Specifies the occurrence time of an event.
Required Attrs.: Infra.Timestamp

ReceptionTime is-a Represent.ComplexSemanticObject
Description: Specifies the reception time of an event notification.
Required Attrs.: Infra.Timestamp
...

Appendix C

Ontology Definition - Domain-specific

C.1 Online-Auction Domain (Auction)

AuctionItem is-a Represent.ComplexSemanticObject
DESCRIPTION: Identifies an item that is being auctioned. This includes the seller identification (*AuctionParticipant*), the item identification (*ItemId*), as well as other information that describes the characteristics of the corresponding auction process.
REQUIRED ATTRS.: Auction.ItemIdentifier, Auction.AuctionParticipant, Auction.ItemHeadline, Auction.ItemCategory, Auction.AuctionStart, Auction.AuctionDeadline, Auction.AuctionType, Auction.StartPrice

AuctionParticipant is-a Represent.CharString
DESCRIPTION: Specifies a online identification (username) of an auction participant. The participant can play the role of bidder and seller.

AuctionDeadline is-a Represent.DateTime
DESCRIPTION: Identifies the end of the auction process.

AuctionSite is-a Represent.CharString
DESCRIPTION: Specifies the name of an auction service provider.

AuctionStart is-a Represent.DateTime
DESCRIPTION: Identifies the begin of an auction process.

AuctionType is-a Represent.CharString
DESCRIPTION: Specifies the kind/method of an auction process.
DOMAIN: {"english", "dutch", "reverse"}

BidAmount is-a Represent.Price
DESCRIPTION: Identifies an amount of money that a bidder offers for an item during its auction process.

StartPrice is-a Represent.Price
DESCRIPTION: Identifies the starting price of an item.

BidTimestamp is-a Infra.Timestamp
DESCRIPTION: Represents the date and time when a bid is placed.

ItemCategory is-a Represent.CharString
DESCRIPTION: Specifies a category of an auction item taxonomy.

ItemDescription is-a Represent.CharString
DESCRIPTION: Represents a description of the item that is being auctioned.

ItemHeadline is-a Represent.CharString
DESCRIPTION: Gives concrete (marketing) information describing the item that is being auctioned.

ItemIdentifier is-a Represent.CharString
DESCRIPTION: Identifies an *AuctionItem*. The identifier is unique only within the context of a given *AuctionSite*.

ItemMinPrice is-a Represent.Price
DESCRIPTION: Specifies the minimum price at which the *AuctionItem* can be effectively sold. Normally this information is maintained hidden.

HighestBid is-a Auction.BidAmount
DESCRIPTION: Specifies the highest price that is being offered.

TimeLeft is-a Represent.TimeAmount
DESCRIPTION: Remaining time period to the end of an auction process.

...
BidderAgent is-a Represent.ComplexSemanticObject
DESCRIPTION: Represents an agent that places bids on behalf of a bidder (*AuctionParticipant*).

AgentLimit is-a Represent.Price
DESCRIPTION: Specifies the maximum limit that the bidder agent can reach when bidding.

REQUIRED ATTRS.: Auction.AuctionParticipant, Auction.ItemIdentifier, Auction.AgentLimit

Events:

AuctionRelatedEvent is-a Infra.Event
DESCRIPTION: Specifies the occurrence of an auction related happening.

ProcessRelated is-a Auction.AuctionRelatedEvent
Description: Identifies events related with the auction process.

BiddingEvent is-a Auction.ProcessRelatedEvent
Description: Identifies notifications related with the ongoing bidding process.

PlaceBid is-a Auction.BiddingEvent
Description: Identifies the placement of a new bid containing the bidder identification (*AuctionParticipant*), the item in question and the amount of money offered.
Required Attrs.: Auction.AuctionParticipant, Auction.ItemIdentifier, Auction.BidAmount

AgentLimitReached is-a Auction.BiddingEvent
Description: Identifies the situation when a bidder agent reaches the *AgentLimit* and is outbidded.
Required Attrs.: Auction.ItemIdentifier, Auction.AuctionParticipant, Auction.AgentLimit, Auction.HighestBid

CurrentHighestBid is-a Auction.BiddingEvent
Description: Specifies that the *AuctionParticipant* who receives this event is the highest bid at the (current) moment. (This event is part of the place bid protocol. It is used to acknowledge the placement of a bid.)
Required Attrs.: Auction.ItemIdentifier, Auction.AuctionParticipant, Auction.HighestBid

NiceTry is-a Auction.BiddingEvent
Description: Specifies that an *AuctionParticipant* has placed a bid but was immediately outbidded. As a consequence he receives this *BiddingEvent*. (This event is part of the place bid protocol. It is used to acknowledge the placement of a bid.)
Required Attrs.: Auction.ItemIdentifier, Auction.AuctionParticipant, Auction.HighestBid

OutBid is-a Auction.BiddingEvent
Description: Specifies that the *AuctionParticipant* who receives this event was outbidded.
Required Attrs.: Auction.ItemIdentifier, Auction.AuctionParticipant, Auction.HighestBid

NewParticipant is-a Auction.BiddingEvent
Description: Identifies that a new *AuctionParticipant* participates (had placed a bid) of an auction process.
Required Attrs.: Auction.ItemIdentifier, Auction.AuctionParticipant

TimeRelated is-a Auction.ProcessRelatedEvent
Description: Identifies happenings of the auction process related with time.

StartOfAuction is-a Auction.TimeRelatedEvent
DESCRIPTION: Identifies the begin of an auction process (*AuctionStart*).
REQUIRED ATTRS.: Auction.ItemIdentifier, Auction.AuctionStart

ApproachingEnd is-a Auction.TimeRelatedEvent
DESCRIPTION: Identifies that the end of an auction process is approaching, (*TimeLeft* to go, could be specified).
REQUIRED ATTRS.: Auction.ItemIdentifier, Auction.AuctionDeadline

ShortedDeadline is-a Auction.TimeRelatedEvent
DESCRIPTION: Specifies that the *AuctionDeadline* was shortened.
REQUIRED ATTRS.: Auction.ItemIdentifier, Auction.AuctionDeadline

ExtendedDeadline is-a Auction.TimeRelatedEvent
DESCRIPTION: Specifies that the *AuctionDeadline* was extended.
REQUIRED ATTRS.: Auction.ItemIdentifier, Auction.AuctionDeadline

EndOfAuction is-a Auction.TimeRelatedEvent
DESCRIPTION: Identifies the end of an auction process (*AuctionDeadline*).
REQUIRED ATTRS.: Auction.ItemIdentifier

ResultEvent is-a Auction.ProcessRelatedEvent
DESCRIPTION: Identifies events related with the end of an auction process.

SoldEvent is-a Auction.ResultEvent
DESCRIPTION: Identifies specific events related with a "normal" end of an auction process.

YouLost is-a Auction.SoldEvent
DESCRIPTION: The *AuctionDeadline* was reached and another *AuctionParticipant* is the winner of the corresponding auction process.
REQUIRED ATTRS.: Auction.AuctionParticipant, Auction.ItemIdentifier

YouWon is-a Auction.SoldEvent
DESCRIPTION: The *AuctionDeadline* was reached and the bidder (*AuctionParticipant*) who receives this notification is the winner of the auction process in question (she placed the highest bid).
REQUIRED ATTRS.: Auction.AuctionParticipant, Auction.ItemIdentifier, Auction.BidAmount

CancellationBySeller is-a Auction.ResultEvent
DESCRIPTION: An auction process was cancelled due to seller's (*AuctionParticipant*) decision.
REQUIRED ATTRS.: Auction.ItemIdentifier

UnderMinPrice is-a Auction.ResultEvent
DESCRIPTION: Identifies the situation where an auction process was ended (*AuctionDeadline* was reached) but the *AuctionMinPrice* was not reached.
REQUIRED ATTRS.: Auction.ItemIdentifier

OfferRelated is-a Auction.AuctionRelatedEvent
Description: Identifies those happenings related with the offering of new items or categories.

NewItemOfInterest is-a Auction.OfferRelatedEvent
Description: Identifies the creation of a new item which matches the characteristics specified by *AuctionParticipant*s that receive this event.
Required Attrs.: Auction.ItemIdentifier, Auction.ItemCategory, ItemHeadline

NewCategory is-a Auction.OfferRelatedEvent
Description: Identifies the creation of new category.
Required Attrs.: Auction.ItemCategory

...

C.2 Car Domain (Car)

CarIdentifier is-a CoolTown.CoolTownId
Description: Represented the identification of a vehicle. According to the underlying CoolTown model, it is represented with a URL that, in this case, points to the car portal.

CarInfo is-a Represent.ComplexSemanticObject
Description: Represents the information related to a particular car, involving information like, *Make*, *TypeOfCar*, *Model*, *Year*, *Color*, etc.
Required Attrs.: Car.CarIdentifier, Car.Make, Car.Model, Car.Year

Make is-a Represent.CharString
Description: Identifies the car manufacturer, e.g. Volkswagen, BMW, Mercedez-Benz, etc.

TypeOfCar is-a Represent.CharString
Description: Identifies the main characteristics of a vehicle, e.g. sedan, convertible, etc.

Model is-a Represent.CharString
Description: Identifies a particular version/model of a vehicle, e.g. Golf, Passat, 315, CLK, etc.

Year is-a Represent.CharString
Description: Represents the year in which a car rolls off the assembly line.

CarStatus is-a Represent.ComplexSemanticObject
Description: Represents the state of a car involving values of sensors and other related information, like, *EngineStatus*, *InTemperature*, *Speedometer*, *Odometer*, *Occupants*, *GeoLocation*, etc.
Required Attrs.: Car.CarIdentifier, Car.EngineStatus, Car.Speedometer, Car.Odometer

EngineStatus is-a Represent.ComplexSemanticObject
DESCRIPTION: Represents the state of an engine which involves values of sensors and other related information, like, *RPM*, *MotorTemperature*, etc.
REQUIRED ATTRS.: Car.CarIdentifier, Car.RPM

MotorTemperature is-a Represent.PhysicalQuantity
DESCRIPTION: Represents the heat inside of the motor measured on a particular scale.

InTemperature is-a Represent.PhysicalQuantity
DESCRIPTION: Represents the heat in the interior of the car. It is measured on a particular temperature scale such as the Fahrenheit or Celsius scale.

OutTemperature is-a Represent.PhysicalQuantity
DESCRIPTION: Represents the heat outside of the car. It is measured on a particular temperature scale such as the Fahrenheit or Celsius scale.

Odometer is-a Represent.PhysicalQuantity
DESCRIPTION: Represents the distance traveled by a car.

Speedometer is-a Represent.PhysicalQuantity
DESCRIPTION: Represents the vehicle's speed.

Occupants is-a CoolTown.SetOfCoolTownIds
DESCRIPTION: Represents the set of persons and mobile devices that are inside of a vehicle at a particular point in time.

FuelLevel is-a Represent.PhysicalQuantity
DESCRIPTION: Represents the amount of fuel inside the tank.

PartId is-a CoolTown.CoolTownId
DESCRIPTION: Represented the identification of a part. According to the underlying CoolTown model a URL is used to point to its portal (web representative).

ProducedBy is-a Car.Manufacturer
DESCRIPTION: Identifies a manufacturer that produced the product in question.

ProductionDate is-a Represent.DateTime
DESCRIPTION: Identifies the date in which the associated product was manufactured.

RPM is-a Represent.PhysicalQuantity
DESCRIPTION: Represents the amount of revolutions per minute of an engine.

...

Events:

CarHappening is-a Infra.PrimitiveEvent
DESCRIPTION: Represents an elementary car-related occurrence of interest.
REQUIRED ATTRS.: Car.CarIdentifier

DriverGetInto	is-a Car.CarHappening
Description:	Represents the happening where the driver gets into the car.
Required Attrs.:	CoolTown.CoolTownId
GetInto	is-a Car.CarHappening
Description:	Represents the happening where the persons and/or objects (e.g PDA) get into the car.
Required Attrs.:	CoolTown.CoolTownId
LowFuelLevel	is-a Car.CarHappening
Description:	Identifies the situation where the level of the fuel is below a predefined level.
Required Attrs.:	Car.FuelLevel
PartFailure	is-a Car.CarHappening
Description:	Represents the occurrence of a failure on particular part of a car.
Required Attrs.:	Car.PartId
NewLocation	is-a Car.CarHappening
Description:	Represents the geographical location of a vehicle obtained, for example, from a GPS receiver.
Required Attrs.:	Car.GeoLocation

...