A Description Framework and Event-Driven Architecture for the Semantic Web and Semantic Grid

A thesis submitted for the degree of

PhD Thesis

Author: Rafael González Cabero
Advisors: Dr. María de los Santos Pérez Hernández
Dr. Asunción de María Gómez Pérez

February 2012
Para todas las personas que me han ayudado a llegar hasta aquí
# Index

## Table of Contents

**Chapter 1** Introduction and Overview ................................................................. 15

1.1 The Semantic Web and the Semantic Grid ......................................................... 16

1.2 Virtual Organizations and Grid Service-Oriention ............................................. 18

1.3 A knowledge level description framework for both Web and Grid Services 19
   1.3.1 From Semantic Web Services to Semantic Grid Services: Minding the Gap .... 19
   1.3.2 From the Symbol Level to the Knowledge Level: Minding the Step .............. 20

1.4 An Event-Driven Architecture for the Semantic Grid and the Semantic Web 21

1.5 Document Structure and Overview .................................................................... 23

**Chapter 2** State of the Art ...................................................................................... 25

2.1 Ontologies for Services Description ................................................................. 25
   2.1.1 OWL-S Ontology .......................................................................................... 28
      2.1.1.1 Service Profile ...................................................................................... 29
      2.1.1.2 Service Model ..................................................................................... 30
      2.1.1.3 Service Grounding .............................................................................. 32
   2.1.2 WSMO ........................................................................................................ 33
      2.1.2.1 WSMO Ontology Main Elements ......................................................... 33
         2.1.2.1.1 Ontologies ..................................................................................... 33
         2.1.2.1.2 Web Services .................................................................................. 34
         2.1.2.1.3 Goals .............................................................................................. 35
         2.1.2.1.4 Mediators ....................................................................................... 35
         2.1.2.1.5 Interfaces ....................................................................................... 36
         2.1.2.1.6 Annotations ................................................................................... 37
   2.1.2.2 WSML ...................................................................................................... 37
   2.1.3 IRS Framework ............................................................................................ 38
   2.1.4 Limitations ................................................................................................... 40

2.2 Semantic Web Services Provisioning Architectures ......................................... 43
   2.2.1 IRS Framework ............................................................................................ 43
      2.2.1.1 IRS Server ........................................................................................... 44
      2.2.1.2 Publishing platforms .......................................................................... 46
      2.2.1.3 IRS Client ............................................................................................ 46
4.3.1.1 Functional Profile .................................................................95
4.3.1.2 Non-Functional Profile .........................................................95
4.3.2 Service Processes .........................................................................98
  4.3.2.1 Orchestration Process .........................................................98
  4.3.2.2 Interaction Process .............................................................99
  4.3.2.3 Interactionflow Patterns ....................................................103
    4.3.2.3.1 Basic Interactionflow Pattern ......................................103
    4.3.2.3.2 Branching and Synchronization Interactionflow Patterns ..........106
4.4 Virtual Organization Model ................................................................109
  4.4.1 Agents Submodel .......................................................................110
  4.4.2 Roles Submodel .........................................................................114
  4.4.3 Events Submodel .........................................................................117
    4.4.3.1 Eventflow Patterns ........................................................121
      4.4.3.1.1 Property Eventflow Patterns ....................................122
      4.4.3.1.2 Structural Eventflow Patterns ...................................123
      4.4.3.1.3 Logical Eventflow Patterns .......................................126
    4.4.4 Choreography Submodel .........................................................127
  4.4.5 Themes Submodel ......................................................................130
  4.4.6 Organizations Submodel ..........................................................133
    4.4.6.1 Policy Patterns ...............................................................135
      4.4.6.1.1 Exclusive Alternatives Policy Pattern .........................136
      4.4.6.1.2 Conjunction Policy Pattern ......................................136
      4.4.6.1.3 Disjunction Policy Pattern .........................................137
4.5 Firenze Framework Design and Annotation Process ............................138

Chapter 5 Vega ......................................................................................141
  5.1 Vega Architectural Principles ........................................................142
    5.1.1 Distributed Knowledge System Paradigm ...............................143
    5.1.2 Service-Oriented Paradigm Principles ....................................144
    5.1.3 Event-driven Paradigm Principles .........................................147
    5.1.4 Autonomic Computing Principles ..........................................148
  5.2 Vega Architecture ..........................................................................150
    5.2.1 Service Bus and Semantic Service Bus ....................................151
    5.2.2 Agent Adapters ......................................................................153
    5.2.3 Design Tools ..........................................................................153
    5.2.4 Event-driven Architecture Modules .......................................156
  5.3 The Event Handling Architecture ..................................................157
5.3.1 The Event Handler in Detail

5.4 The Event Brokering Architecture

5.4.1 Model for Expertise-Based Event Routing

5.4.1.1 Peers

5.4.1.2 Shared Model

5.4.1.3 Expertise

5.4.1.4 Notification

5.4.1.5 Subject

5.4.1.6 Abstraction Process

5.4.1.7 Matching Process

5.4.2 Expertise Knowledge Representation and Creation

5.4.2.1 Types of Expertise

5.4.2.1.1 According to Purpose

5.4.2.1.2 According to the Peer Type

5.4.2.2 Binary Decision Diagrams for Expertise Representation

5.4.2.2.1 Binary Decision Diagrams for Representing Event Handlers Expertise

5.4.2.2.2 High-level Binary Decision Diagram for Representing Expertise

5.4.2.3 Expertise Descriptions Representation

5.4.2.3.1 Local Brokers Expertise Representation

5.4.2.3.2 Border and Inner Brokers Expertise Representation

5.4.2.4 Expertise Representation Creation

5.4.2.4.1 Event Handler Expertise Representation Creation

5.4.2.4.2 Brokers Expertise Representation Creation

5.4.3 The Expertise-Based Routing Algorithm

5.4.3.1 The Abstraction Process

5.4.3.2 The Matching Process

5.4.3.2.1 Excessive number of neighbours

5.4.3.2.2 Length of the expertise formulae in the entries

5.4.3.2.3 The complexity associated with evaluating each of the nodes

5.5 The Event Detection Architecture

5.5.1 The Event Detector in Detail

5.5.2 Eventflow Knowledge Representation and Creation

5.5.2.1 Complex Events Detection

5.5.2.1.1 Property Related Patterns

5.5.2.1.2 Structural Eventflow Patterns

5.5.2.1.3 Iteration Eventflow Pattern

5.5.2.1.4 Logical Eventflow Patterns
Table of Figures

Figure 1-1 Semantic Grid relationship with other activities. ................................................................. 16
Figure 1-2 Representation levels pyramid ............................................................................................. 20
Figure 1-3 Abstract representation the anatomy of an event-driven architecture (adapted from (Mühl et al., 2006)). ......................................................................................................................... 21
Figure 2-1 OWL-S upper concepts and ontologies diagram (adapted from (OWL-S, 2004)) ................. 29
Figure 2-2 Service Profile Ontology main elements (adapted from (OWL-S, 2004)) ................................. 30
Figure 2-3 OWL-S Process description elements diagram (adapted from (OWL-S, 2004)) ................. 30
Figure 2-4 Different types of OWL-S processes and its components (adapted from (OWL-S, 2004)) ....... 31
Figure 2-5 WSMO Ontology Top-Level Elements (adapted from (WSMO, 2004)) ................................. 33
Figure 2-6 Web service description components (adapted from (WSMO, 2004)) ................................. 34
Figure 2-7 Goal description components (adapted from (WSMO, 2004)) ........................................... 35
Figure 2-8 WSML variations diagram (WSML, 2005) ............................................................................. 38
Figure 2-9 IRS-II Framework main knowledge elements (adapted from (Motta et al., 2003)) ............... 39
Figure 2-10 Representation of different description frameworks regarding complexity and design level. .............................................................................................................................................. 42
Figure 2-11 IRS Framework software architecture (adapted from (Motta et al., 2003)) ......................... 44
Figure 2-12 WSMX architecture (adapted from (WSMXb, 2005)) ......................................................... 48
Figure 2-13 Interaction modes table (adapted from (Mühl et al., 2006)) .............................................. 50
Figure 2-14 Taxonomy of distributed event-based systems (adapted from (Tarkoma and Raatikainen, 2006)) ................................................................................................................................. 51
Figure 2-15 Content-based routing brokering step complexity sources .............................................. 57
Figure 4-1 Firenze Framework as a layered description framework ...................................................... 69
Figure 4-2 Problem-Solving Methods Model main components .............................................................. 73
Figure 4-3 Anatomy of a Problem-Solving Method and its perspectives .............................................. 74
Figure 4-4 Task description main concepts and relationships ............................................................... 75
Figure 4-5 Methods description main concepts and relationships ......................................................... 77
Figure 4-6 The inputs and outputs of a complex method in relation with its knowledgeflow ................. 78
Figure 4-7 Controflow patterns catalogue ............................................................................................ 79
Figure 4-8 Sequence controflow pattern graphical representation ....................................................... 80
Figure 4-9 Parallel controflow patterns graphical representation ....................................................... 80
Figure 4-10 Synchronization controflow pattern graphical representation ........................................ 81
Figure 4-11 Exclusive choice controflow pattern graphical representation ......................................... 81
Figure 4-12 Simple merge controflow pattern graphical representation ............................................ 82
Figure 4-13 Multi-choice controflow pattern graphical representation ............................................. 83
Figure 5-20 Rationale behind the proposed routing algorithm ................................................................. 189
Figure 5-21 The abstraction process ........................................................................................................ 190
Figure 5-22 The matching process ........................................................................................................... 191
Figure 5-23 The three axes of complexity of the matching process ......................................................... 192
Figure 5-24 The concept of blurred expertise and the matching function .............................................. 194
Figure 5-25 The event detection architecture ............................................................................................ 199
Figure 5-26 Event handler module main components ................................................................................. 200
Figure 5-27 Detail of the event detector input mailbox .............................................................................. 201
Figure 5-28 The levels of representation pyramid revisited (II) ............................................................... 202
Figure 5-29 Abstract state machine for the detection of an event ............................................................ 204
Figure 5-30 Evolution of an abstract state machine into two different ones ........................................... 204
Figure 5-31 Abstract state machine for the property relater eventflow pattern .................................. 205
Figure 5-32 Abstract state machine for the detection of the sequence eventflow pattern .................. 206
Figure 5-33 Abstract state machine for the detection of the causality eventflow pattern .................... 207
Figure 5-34 Abstract state machine for the detection of the iteration eventflow pattern .................... 208
Figure 5-35 Abstract state machine for the detection of the conjunction/disjunction eventflow patterns .......................................................... 209
Figure 5-36 Abstract state machine for the detection of the negation eventflow patterns ................. 210
Figure 6-1 The different uses across time of the Firenze Framework ......................................................... 216
Figure 6-2 ODESWS description process ................................................................................................. 217
Figure 6-3 ODESGS description and annotation process ..................................................................... 219
Figure 6-4 Using Problem-Solving Methods for interpreting logs ......................................................... 221
Figure 6-5 From abstract processes to concrete executable processes ............................................... 223
Figure 6-6 Parametric design search, and its multi-agent extension ....................................................... 224
Figure 6-7 Process templates as a Problem-Solving Method method controlflow .................................. 225
Figure 6-8 Graphical representation of the generic service service ...................................................... 230
Figure 6-9 Announce capability task graphical representation .............................................................. 230
Figure 6-10 Simple brokering topology ................................................................................................. 232
Figure 6-11 Best-case scenario response times graphical representation ............................................. 235
Figure 6-12 Worst-case scenario graphical representation ..................................................................... 237
Figure 6-13 Average times graphical representation .............................................................................. 238
Figure 6-14 Service-oriented vs event-driven architectures (adapted from (Kazi, 2005)) .................... 240
Figure 8-1 Ontology concept main elements ........................................................................................... 250
Figure 8-2 Term relation graphical representation ................................................................................. 252
Figure 8-3 Taxonomic relations among concepts ................................................................................. 253
Figure 8-4 Is-A Graphical representation ............................................................................................... 253
Figure 8-5 A binary decision diagram that represents the Boolean function A .................................. 257
Figure 8-6 Taxonomy of Abstract State Machines' ............................................................................. 260
Figure 8-7 Taxonomy of functions (adapted from (Börger and Stärk, 2003)).

Figure 8-8 An abstract state machine as an automata.

Figure 8-9 Expertise Based Matching.
Chapter 1  Introduction and Overview

The nature of the work presented in this thesis is twofold. On the one hand, we present the Firenze Framework, a knowledge level description framework rich enough to express the semantics required for describing services, concretely Semantic Web services and Semantic Grid services. On the other hand, we propose Vega, an event-driven architecture that, by means of proposed knowledge level description framework, is able to achieve high scale provisioning of knowledge-intensive services. In this chapter we put this work in context, presenting some of the main concepts and rationale that will be present along the rest of the dissertation.

In this introductory chapter we put in context and give a brief outline of the work that we thoroughly present in the rest of the dissertation. We consider this work divided in two main parts. The first part is the Firenze Framework, a knowledge level description framework rich enough to express the semantics required for describing both semantic Web services and semantic Grid services.

We start by defining what the Semantic Grid is and its relation with the Semantic Web; and the possibility of their convergence since both initiatives have become mainly service-oriented. We also introduce the main motivators of the creation of this framework, one is to provide a valid description framework that works at knowledge level; the other to provide a description framework that takes into account the characteristics of Grid services in order to be able to describe them properly.

The other part of the dissertation is devoted to Vega, an event-driven architecture that, by means of proposed knowledge level description framework, is able to achieve high scale provisioning of knowledge-intensive services. In this introductory chapter we portrait the anatomy of a generic event-driven architecture, and we briefly enumerate their main characteristics, which are the reason that make them our choice.

How happy is the blameless vestal's lot!
The world forgetting, by the world forgot.
Eternal sunshine of the spotless mind!
Each pray'r accepted, and each wish resign'd.

Alexander Pope
1.1 THE SEMANTIC WEB AND THE SEMANTIC GRID

The Semantic Grid is the result of the convergence of the Semantic Web and Grid technologies. Its definition is obtained modifying the Semantic Web definition given in (Hendler, 2001). The Semantic Grid is defined thus as an extension of the current Grid, in which information and services are given well-defined meaning for better enabling computers and people to work in cooperation. Figure 1-1 relates different technologies in terms of scale of interoperability versus scale of data and computation. It is important to note that they are not orthogonal activities, but they complement and benefit each other. Henceforth, depending on our requirements we can place ourselves in different coordinates or walk different paths (represented as dotted lines in Figure 1-1).

Figure 1-1 Semantic Grid relationship with other activities.

The Semantic Grid vision implies a set of the requirements and research challenges. These requirements were initially identified in an unimpeachable manner in (De Roure et al., 2001); and have been updated in (De Roure et al., 2005) (NGG3, 2006). The most closely related with the presented work are the following:

- **Ontologies and content annotation.** Contents in the system (resources, services, provenance data, etc.) have associated meta-data. All this meta-data information should be expressed in terms of ontologies, which provide agreed formal interpretations.

- **Level of abstraction raise.** As exposed in (NGG3, 2006) there is a necessity in future generation Grid systems to raise the level of abstraction at all the levels. Users, programmers, and system administrators should be provided with higher level programming models and tools, as well as with better management abstractions.

---

1For further details about the Semantic Grid vision and activities visit http://www.semantigrid.org
2 Adapted from the figure that was presented in http://www.semanticgrid.org/vision.html
o **Resource and service descriptions discovery and use.** Grid computing implies the handling of enormous volumes of distributed content, either static or dynamic. Thus in a Semantic Grid environment we should assume the existence of a huge number of services instances and resources descriptions, which should be efficiently discovered in order to be used in an effective way.

o **Process description, coordination and enactment.** Complex services descriptions must be bestowed to provide mechanisms for the (semi) automatic composition of services, the achievement of distributed enacting, and the coordination of their activities.

o **Autonomic and proactive behaviour.** In order to cope with the constant evolution of services and resources in a Grid environment, systems should at some extent auto-configure. Following the self-* paradigm. (Kephart and Chess, 2003) Grid systems should exhibit self-configuration, self-management, self-optimization, and self-healing capabilities.

o **Information integration.** The possibility of formulating meaningful queries over disparate information stores can be benefited by the definition of ontologies. They provide a shared vocabulary of the elements of a concrete domain. Thus they facilitate the definition of relationships among elements of different domain ontologies in the form of mappings. As pointed out in (De Roure et al., 2005) this is a classical scenario where the Semantic Web technologies come to play.

o **Ease of configuration and deployment.** Actual Grid environments are hard to configure and deploy. Most of the configuration knowledge is embedded in tedious configuration files; even in some cases configuration is made programmatically. Also the expertise needed to setup the environments is considerable and the process is slow, costly and error prone (all of which reduce the responsiveness of the environment). This requirement is closely related with the quest for self-management that we have already mentioned.

o **Communities.** Users (especially system administrators) should be able to form, maintain and disband Virtual Organizations (Foster et al., 2002). The concept of Virtual Organization will be described in the following section, but it should be stressed that its (semi) automated creation and dissolution requires the formal description of their membership criteria, the rules for operating inside the community, and the entities to be shared.
1.2 Virtual Organizations and Grid Service-Orientation

Grid computing is about sharing and combining heterogeneous computational resources across different organizational boundaries aiming either at a higher goal or at reusing idle resources; the result sum of such computational resources constitutes an institution named Virtual Organization. This key concept in Grid computing was initially presented in (Foster et al., 2001) as a coalition of disperse organizations and/or individuals that share resources in a controlled fashion. This definition was extended by (Dimitrakos et al., 2004) where Virtual Organizations became categorized as temporary or permanent coalitions of geographically dispersed individuals, groups, organizational units or entire organizations that pool resources, capabilities and information to achieve common objectives.

The initial definition of Virtual Organization evolved, as Grid computing became service-oriented. Service-orientation is a design paradigm whose main principle is the separation of concerns, which aims at fostering following design principles (Erl, 2007): standard service contract, service loose coupling, service abstraction, service reusability, service statelessness, service discoverability, and service composeability. Grid computing became service-oriented since the emergence of OGSA (Open Grid Service Architecture) (Foster et al., 2002). Grid resources were wrapped with services and exposed via a set of operations specified in a standard XML language. OGSA thus redefined the concept of Virtual Organization, becoming them the set of services that the participant organizations and/or individuals operate on and share (Foster et al., 2002) (plus some sharing policies). This idea of service-oriented Virtual Organization, mixed with an agent-oriented and dynamic view, was further described in (Foster et al., 2004); in that paper Virtual Organizations were considered dynamic agents marketplaces, a form of dynamic service composition where a number of initially distinct entities come together, under a set of operating conditions, to form a new entity that offers a new service (Foster et al., 2004).

Finally, Grid service-orientation transition became almost definitive since widespread use of service-oriented Grid computing environments and implementation technologies, such as GT4 and WSRF (Czajkowski et al., 2003). Grid environments are now compliant with the most widely accepted Web services standards and technologies (WSDL, Christensen et al., 2001), SOAP (Box et al., 2000), etc.).

---

3 http://www.globus.org/toolkit
1.3 A KNOWLEDGE LEVEL DESCRIPTION FRAMEWORK FOR BOTH WEB AND GRID SERVICES

In this work, we present a knowledge-level description framework for describing both Web services and Grid services. The easiest approach might seem the adoption and reuse of all the emerging technologies related to Semantic Web services (i.e. IRS (Motta et al., 2003)(Domingue et al., 2008a), OWL-S (OWL, 2004), WSMO (WSMO, 2004), WSDL-S (Akkiraju et al., 2005), etc.). Thought, these technologies and initiatives should not be considered off-the-shelf solutions, as we should take into account:

- **The different nature of a Web service and a Grid service.** Grid services are different from Web services. As a consequence their formal descriptions are likely to be different. We will depict this topic in section 1.3.1 From Semantic Web Services to Semantic Grid Services: Minding the Gap.

- **The level of representation of current initiatives.** Current service description initiatives work at the symbol level. This characteristic makes their utilization more prone to inconsistencies and errors; and may affect the quality of the design since it is constrained by the characteristics of the chosen language. We will devote section 1.3.2 From the Symbol Level to the Knowledge Level: Minding the Step to that topic.

As a consequence in this thesis we propose a Grid aware framework for the knowledge level description of services able to formalize both Web and Grid services.

1.3.1 From Semantic Web Services to Semantic Grid Services: Minding the Gap

As we have stated, one of the main points for the convergence of the Semantic Web and the Semantic Grid may lie in their service-oriented view. But before reusing the Semantic Web services technology in the Grid, we should analyze the different nature of these entities, Grid services and Web services. In (Goble et al., 2005b) we performed this analysis, identifying the main differences between Web and Grid services. We concluded on the necessity of a description framework that includes the formalization of Virtual Organization; a deeper approach to annotation of non-functional properties; complex interactions, including several participants and several flows of interactions; and a richer description model for services, since all existing frameworks have taken a transformational approach. We will describe these necessary extensions in Chapter 4 Firenze Framework.
1.3.2 From the Symbol Level to the Knowledge Level: Minding the Step

In order to cope with the new Grid requirements the shortest path seems to adapt one of the enumerated service description initiatives. We do not follow such approach since all of them work at the symbol level and we foresee that a higher level of abstraction would bring many benefits. The Knowledge Level concept was introduced by Allan Newell in (Newell, 1982). The knowledge level hypothesis, as depicted in Figure 1-2, postulates that there exists a distinct computer systems level, lying immediately above the symbol level, which is characterized by knowledge as the medium and the principle of rationality as the law of behaviour (Newell, 1982). There were many approaches that took Newell’s postulates as a starting point such as Generic Task Approach (Chandrasekaran, 1986), Role-Limiting Methods (McDermott, 1988) KADS (Wielinga et al., 1992), and finally CommonKADS (Schreiber et al, 1999). Our framework, based partly on concepts of CommonKADS, will be described later, but in this introduction we want to summarize the advantage of describing systems at the knowledge level. Knowledge level descriptions permit to make meaningful statements about the system behaviour without reference to the structures and reasoning mechanism. Thus, designs at the knowledge level exhibit the following properties:

- **Designs made at the knowledge level are reusable and shareable.** As we describe later on in this thesis we can establish different relationships between a knowledge level description and several representations generated from it.

- **Designs made at knowledge level effectively raise the level of abstraction of services descriptions.** Current initiatives handle directly representational languages, which is more prone to inconsistencies and errors. Furthermore the quality of the design is leveraged due to the possible limitation imposed by the target low-level language. This is a direct

![Figure 1-2 Representation levels pyramid.](image-url)
Introduction and Overview

consequence of breaking the separation between the design and implementation phases, which is not the case in the framework that we propose.

All these design properties justify on its own the creation of a knowledge level framework, as on the one hand allows the creation of high-level designs that are more reusable and technically sounder; and on that also contains all the new elements necessary to describe both Grid services and Web services. We call this description framework the Firenze Framework, and it will be depicted thoroughly in the Chapter 4 Firenze Framework.

1.4 AN EVENT-DRIVEN ARCHITECTURE FOR THE SEMANTIC GRID AND THE SEMANTIC WEB

Parallel to the described Semantic Web and Semantic Grid initiatives, the event-based systems field has been gaining momentum. The characteristic that makes event-based systems unique is their model of interaction. As described in (Cazaniga et al., 1998), the main novelty and innovation of the event-based architectural style is that neither the published notifications nor the subscriptions are directed toward specific components. The producer of an event notification is unaware of the possible consumers; the event-based system is the one responsible of delivering these notifications to the right consumer modules.

Figure 1-3 Abstract representation the anatomy of an event-driven architecture (adapted from (Mühl et al., 2006)).

Figure 1-3 represents the anatomy of a generic event-driven system that makes possible such interaction model. A set of domain dependent component are grouped around a notification service and effectively connected by a communication infrastructure. More precisely:

○ The notification service acts as a mediator (see the Mediator Pattern in (Gamma et al., 1995)). Consumers manifest interest in receiving a kind of event. The notification service using this information is responsible of handing over notifications to the right components; it must deliver every published notification to all the interested consumers. The producer of
the event is not aware of which are the notification recipients; it just informs of some happening of interest to the system. The notification service presented in Figure 1-3 is an abstraction, since it might range from a single monolithic broker to a complex net of distributed event routers. From a functional perspective there is no difference, but most of the non-functional properties of the event-based system (like scalability, efficiency, latency etc.) depend upon the architecture chosen for the notification service.

- The **communication infrastructure** is included to abstract the rest of the system from the underlying communication technology. One of the common misunderstandings about event-driven systems is to consider them as synonym of message-passing systems. Albeit event-based systems are usually implemented using message-passing technology, they are different concepts. Also notice that there are other alternatives for implementing event-based systems, such as those based on tuplespaces (Gelernter et al., 1985). In this thesis we propose the use of a service bus because of its characteristics, not because it is the only possible choice.

The main benefits of adopting event-driven architectures are identified in (Luckham, 2002)(Mühl et al., 2006)(Hohpe, 2006), and they can be summarized stating that event-based systems carry the potential for easy integration of autonomous, heterogeneous components into complex systems that are easy to evolve and scale (Bates et al., 1998)(Mühl et al., 2006). In the presented work we propose applying the precepts of event-based systems to a semantically described service-oriented architecture, in order to obtain a high-scale semantic services architecture. Current approaches (Benjamins et al., 1998)(Motta et al., 2003)(Haller et al., 2005) are based on the traditional request/reply model of interaction. Because of that they suffer from several limitations that can affect to some of their properties (e.g. high coupling between components, issues with scalability, reactivity problems, among others; as described in (Gore et al., 2001)(Hohpe, 2006)(Kayne, 2003)). Event-driven systems to the contrary posses certain characteristics that make them perfect candidates for sustaining both Web and Grid infrastructures. Briefly they are:

- **Architecture components low coupling.** Current approaches use request/reply mode of interaction, what cause the apparition of many undesired dependencies among the components of the environment. Concisely, the most important, according to their cause are:
  
  • **Synchronous communication related.** A client must wait until the server finishes processing its request and returns the results. Thus, there exist bidirectional coupling between the invoker and the server, because of their lifetimes. Both of them should coexist and be mutually available while the server is processing the request.
Introduction and Overview

- Request/Reply of communication related. Interactions in the actual initiatives introduce unidirectional coupling since invocation is typically targeted at a single operation on a particular service (see (Kayne, 2003) for a thorough dissertation on the topic). The proposed architecture removes this dependence, since the producer does not need to know the details of the consumer interface, moreover, the producer of the event notification does not even know the identity of the consumer or consumers that receive the notification.

- System reactivity. Event-based systems exhibit a huge reacting capacity, even in the case of sparse systems. This is partly due to the handling of fine grained of events in the system and its surrounding environment; but more significantly, because of the possibility of defining and detecting composite events that signify states of interest in the system.

- System scalability. Event-based systems are well known to adapt to growing amounts of work, owing to its capacity of adding and replicating event consumers. Other advantage of distributed event-driven systems is that they are sparse; there is no monolithic server that has to store the session state of several service operation requests.

In consequence, we believe that the event-driven approach is a good approach for building a truly distributed knowledge-based architecture. We aim at the scalable provisioning of semantic services; to achieve so we propose to move to an event-based mode of interaction plus the use of a set of intelligent peer-to-peer brokers that use the semantic descriptions of components to deliver events notifications. We call this architecture Vega, and will be thoroughly described in Chapter 5 Vega.

1.5 DOCUMENT STRUCTURE AND OVERVIEW

The presented document is structured as follows:

- Chapter 2 State of the Art. This chapter presents the state of the art on the frameworks and architectures that are relevant for the work presented in this thesis. First of all, we introduce a basic description of the motivation for annotating in a formal fashion services; and which are the different types of services that we wish to annotate. After that we describe with some detail the two main ontologies for services descriptions. After introducing the most important description frameworks, we analyze the main semantic Web services provisioning architectures with great detail. Finally, we include a description of the main event-driven architectures, focusing on those that provide a distributed notification system.
Introduction and Overview

- **Chapter 3 Work Objectives.** This chapter presents the goals of this thesis, together with the open research problems that we aim to solve. Besides, we detail the contributions to the current state of the art, and the work hypotheses, the assumptions considered as a starting point for this work and the restrictions of the results presented.

- **Chapter 4 Firenze Framework.** In this chapter we present the Firenze Framework, a knowledge level description framework rich enough to express the semantics required for describing services (both semantic Web and semantic Grid services) and Virtual Organizations. This description framework provides ontologies and representation models to express the semantics required to improve services discovery and (semi) automatic composition; and Virtual Organizations design and handling. Contrary to other description frameworks, its descriptions are made at knowledge level, allowing the creation of high-level reusable designs.

- **Chapter 5 Vega.** This chapter is devoted to Vega, the event-driven architecture and the second main outcome of this thesis. We will describe this truly distributed knowledge-based architecture able to achieve high scale semantic services provisioning. Its main characteristics are its inherent distributed nature and the change of mode of interaction. A set of intelligent peer-to-peer brokers are in charge of handling the notification decentralized backbone bus, around which a set of semantically annotated event-driven services are situated.

- **Chapter 6 Evaluation.** This chapter is devoted to the evaluation of the work presented in this document. We propose two different evaluation procedures made to measure and validate the contributions of this thesis depending on the nature of the evaluated work. For the description framework we depict its usage in different research projects, in which it has been used for the description of the different facets of Web and Grid services. The event-driven architecture also includes an empirically tested to check how the algorithm that we propose behave in terms of scalability and self-manageability.

- **Chapter 7 Conclusions and Future Work.** This chapter contains the conclusions of our work, focusing on the main advances to the state of the art on service description frameworks and semantic services provisioning architectures. In this chapter, we also identify open research issues in both areas.
Chapter 2  State of the Art

This chapter presents a summarized state of the art on the frameworks and architectures that are relevant for the work presented in this thesis. First of all, we introduce a basic description of the motivation for annotating in a formal fashion services; and which are the different types of services that we wish to annotate. After that we describe with some detail the two main ontologies for services descriptions, namely OWL-S and WSMO. After introducing the most important description frameworks, we analyze the main semantic Web services provisioning platforms. We describe the main infrastructures for semantic services provisioning, IRS and WSMX, which are described with a greater detail. Both of them provision WSMO services; but as we shall see, they are quite different both in how they implement WSMO descriptions and in the structure of their architectures. Finally, we present the main distributed event-driven architectures, focusing on how they represent and route event notifications. As we wish to incorporate the principles and algorithms of such architectures in the provisioning of semantic services, we must analyze if they can be used off-the-shelf or we must to some extent extend them.

2.1  ONTOLOGIES FOR SERVICES DESCRIPTION

A service in a broad sense is a collection of functional capabilities, which are offered remotely by a set of network accessible operations. From this rather generic definition we can derive many more specific definitions of services, depending on the parameters and characteristics considered in the classification. Nevertheless, in order to delimit clearly the services that we aim to describe in the Firenze Framework we focus in two main services categorizations: functional categorization and technological categorization.

- **According to its functionality.** (Erl, 2007) classifies services in three main categories, namely Task services, Utility services and Entity services. Task services are those services that perform a concrete business process within the domain; utility services are services that do not provide a domain-concrete added value, but are infrastructural, providing cross-cutting functionality such as event logging, notification, exceptional situation handling, etc; and entity services, which wrap the relevant business entities of a given domain (think of them as wrappers of passive resources that need a service that virtualize them). In (Cohen,
Work Objectives

2007) the classification of services is even more detailed, and it is explicitly regarded as a service ontology. Cohen defines initially two service categories: application services as those that provide application specific functionality, and bus services as infrastructure services.

- **Application services** can be decomposed in capability/activity services, services that implement the business-level capabilities of the organization, and represent the action-centric building blocks that are orchestrated by process services. Finally, as in the case of the Erl’s classification, entity services play the role of domain resources wrappers.

- **Bus services**, depending on the nature of the infrastructure that they provide, can be further divided in the communication services transport messages without being concerned with the content of the messages; and like in the case of (Erl, 2007) utility services that deal with the rest of generic, application-agnostic functionality (e.g. transactional support, logging, monitoring and management, service discovery, etc.).

  - From a technological and implementation related point of view there can be as many as different technologies or implementation paradigms (e.g. Cloud services, dataspaces services, RESTful services, etc.). In the context of this thesis we will focus in Web services and Grid services.

A Web service is an interface that describes a collection of operations that are network-accessible through standardized Web protocols whose features are described using a standard XML-based language (Kreger, 2001)(Curbera et al., 2001). Although there are other ways of defining a Web service, in this thesis we adopt the aforementioned definition because it is the one that best captures the interface nature of what a Web service is (and where its benefits come from). Web services should not be considered software modules or components, since such definitions would break the low coupling principle that initially motivated the creation of Web services. In short, it is not the components; it is the interfaces (Kayne, 2003).

Semantic Web services are the mark-up of Web services that make them computer-interpretable, use-apparent and agent-ready (McIlraith et al., 2001). This formal mark-up of services should enable the following tasks:

---

4 Perhaps the name “bus services” is not a very good name choice, since it associates infrastructural services with a concrete type of infrastructure (i.e. buses); association that in many cases does not hold.
o **Automatic Web service discovery.** Automatic Web service discovery is the automated process of locating services that provide a particular class of service capabilities, while adhering to some client-specified constraints. Currently, this task must be performed by a human who might use a search engine to find a service, read the Web service description page, and check manually (probably after analyzing different invocations of the service) whether it satisfies the desired constraints.

o **Automatic Web service invocation.** The invocation of Web services is usually hard-coded in client applications. On the opposite, automatic service invocation involves bringing the possibility to agents (either human or software) of invoking services just possessing its declarative formal description. Needless to say, with no human in the loop.

o **Automatic Web service composition and mediation.** This task involves the automatic selection, composition, and interoperation of services to perform some complex tasks given a high-level objective description.

Nevertheless, (McIlraith et al., 2001) definition of semantic Web service raises a subtle but important question. Should semantic Web services be constrained with all the characteristics and limitations that the Web services definition imposes (i.e. stateless interfaces, XML compliant, etc.)? Depending on our answer to this question we distinguish between semantic (Web services) or (semantic Web) services.

o **Semantic (Web service).** A semantic (Web service) retains all the characteristics of a Web service, adding just semantic annotations to its domain, its inputs and outputs, and describing its functional properties (precondition, postconditions, etc.). However, it does not portray the internal process or structure of the service, as it remains being an interface. It is just a WSDL file plus some semantics (a clear example of this is WSDL-S (Akkiraju et al., 2005) later SAWSDL (SAWSDL, 2007)). Semantic (Web services) have the advantage of being a direct upgrade from current technology to a semantically enhanced one, which make easier for developers to reuse existing infrastructures. We do not describe them since they are not service description frameworks, but a mechanism for associating WSDL and XML Schema documents with semantic descriptions implemented in RDF.

o **(Semantic Web) service.** (Semantic Web) services are not constrained by the nature of a Web service. It can be a Web service, an agent or anything that provides a service-oriented functionality in the context of the Semantic Web. The description of (Semantic Web) services goes far beyond the idea of an interface since we may find internal reasoning process descriptions, explicit lifecycle, state handling, and many other elements. Therefore, they can be considered a superset of semantic (Web services).
Current semantic Web services description frameworks are clearly closer to the idea of (semantic Web) services since most of them describe, at least, the internal structure of complex Semantic Web services (thus, they fall outside the semantic description of a simple network accessible interface).

Once we have briefly defined Web services and Semantic Web services, let us analyze what Grid services are. As we stated in the service-oriented view of the Grid appeared in OGSA (Foster et al., 2002), a Grid service is defined as a network-enabled entity that provides some capability. Grid services serve to achieve the virtualization (i.e. encapsulation independent of the implementation of physical resources such computational resources, storage resources, networks, programs, data-bases, etc.) of the shared resources.

Thus, by analogy with the aforementioned definition of Semantic Web services (McIlraith et al., 2001), a Semantic Grid service is the mark-up of a Grid service that makes it computer-interpretable, user-apparent and agent-ready. Note that due to the more generic definition of what a Grid service is, we are less constrained in the mark-up of a Grid service than in the mark-up of a Web service (keeping in mind the semantic (Web service) and (Semantic Web) service differentiation that we stated above). A Grid service is not defined as an interface, which makes a big conceptual difference.

In the following sections we present the three major initiatives for describing semantic Web services: OWL-S, IRS Framework and WSMO. It is important to remark that these semantic Web services ontologies act as knowledge representation models; they are not taxonomies of services tailored following a specific criterion. Instead, they provide a set of knowledge components that describe different facets of services. The domain dependent description of a service is made using specific domain ontologies on each specific domain where these frameworks might be applied.

### 2.1.1 OWL-S Ontology

OWL-S (OWL-S, 2004) specification is a joint effort from BBN Technologies, Carnegie Mellon University, Nokia, Stanford University, SRI International and Yale University to define an ontology for semantic markup of Web services. OWL-S is the evolution of DAML-S (Ankolenkar et al., 2002), part of the DARPA Agent Markup Language program.

OWL-S describes services in a semantic manner, using a service ontology implemented in OWL (OWL, 2004) (preferably OWL-DL); in combination with WSDL and SOAP. In consequence OWL-S is heavily coupled with Web services technologies, which as described in (Sollazo et al., 2002), is made to achieve the desirable combination of Web services standard
languages and semantic annotation to make possible the use current Web services infrastructures.

![Diagram of OWL-S upper concepts and ontologies](image)

Figure 2-1 OWL-S upper concepts and ontologies diagram (adapted from (OWL-S, 2004)).

Initially, as shown in Figure 2-1, every description of a Semantic Web service in OWL-S is an instance of the class `Service`. OWL-S defines a set of ontologies, namely Service Grounding, Service Profile, and Service Model ontologies (we represent their top-level elements in Figure 2-1). Each of them covers a different aspect of the Semantic Web service description. This concept serves as an organizational point of reference. It relates what the service does (`Service Profile`, described in section 2.1.1.1 Service Profile); how the service achieves its functionality (`Service Model`, described in section 2.1.1.2 Service Model); and finally, how the formalized service can actually be invoked (`Service Grounding`, described in section 2.1.1.3 Service Grounding). Let us describe each of them in the following sections.

### 2.1.1.1 Service Profile

The `service profile` is the specification of what functionality the service provides and the specification of the conditions that must be satisfied for a successful result (OWL-S, 2004). It provides the information needed for an agent to discover the service. The ontology that contains the concepts and relations that model the profile is the service profile ontology, which main elements we show in Figure 2-2. Its central concept is the `Service Profile`. It embodies a high-level specification of what the service does. It contains the competence of the service, which are its inputs, outputs, preconditions, results and effects (we will describe them in the service Model Ontology); and non-functional properties (mainly human and e-commerce oriented properties). The later properties are specified by means of datatype attributes of the Profile concept and instances of `Service Parameter`.

As shown in Figure 2-2 the `Service Profile` is also related to one or several `Service Category` concepts by means of the `hasCategory` relationship. `Service Category` instances describe categories of services on the basis of some classification, preferably a standard.
2.1.1.2 Service Model

The service model describes how the service achieves its functionality, including the detailed description of its constituent processes. More precisely, the service model tells a client how to use the service, by detailing the semantic content of requests, the conditions under which particular outcomes will occur, and, where necessary, the step by step processes leading to those outcomes. It describes how to ask for the service and what happens when the service is executed (OWL-S, 2004). The formal description of the service model enables several tasks, including the analysis of whether a service meets the needs of the invoker, the automatic composition of semantic Web services, intelligent monitoring of the execution of services, etc.

The service model is specified using processes. A process is a specification of the ways a client may interact with the service; it is mean to be an executable program.
outputs, the information that the process provides to the requester; and any number of local variables. A process also has a number of preconditions, which represent the circumstances that must hold so that the process can be successfully invoked. Finally, the process can have any number of results to specify what is the expected value of the output variables and the changes in the state of the world that the service may cause. Both outputs and effects can be made conditional (i.e. depend on conditions that hold true of the world state at the time the process is performed).

Figure 2-4 Different types of OWL-S processes and its components (adapted from (OWL-S, 2004)).

OWL-S specification distinguishes between three types of processes, namely atomic processes, simple processes, and composite processes.

- **Atomic processes.** Atomic processes correspond to the actions a service can perform by engaging it in a single interaction (OWL-S, 2004); a description of a single step invocable process. In other words, it is an invocation of an OWL-S described service. It expects one (possibly complex) message and returns one (possibly complex) message. Thus an atomic process may have several inputs (or several outputs), but they must be packaged in the same input (or output) message. Atomic processes are associated with a grounding, as we will describe later.

- **Composite processes.** Composite processes specify actions that require multi-step protocols and/or multiple server actions (OWL-S, 2004). Composite processes are not atomic; they maintain some state information while performing. Their descriptions are made composing
Work Objectives

other (composite or non-composite) processes; which are orchestrated using control constructs. The allowed control constructs are shown in Figure 2-4. No formal operational semantics are defined for these constructs, they are just concepts represented in OWL. Briefly they are:

- **Sequence**, which states that a set of control constructs should be carried out in order.
- **Split and Split+Join**, which are constructs that define the parallel execution of several branches of executions, in the case of the Split+Join construct also involves the synchronization of these branches.
- **Any-Order**, a construct that states that the process components can be executed in any order, but sequentially (i.e. tasks can not run in parallel).
- **If-Then-Else**, a construct that let us choose the execution of one or other execution branch depending on a condition.
- **Iterate**, used to define a loop, but making no assumption about how many iterations are made.
- **Repeat-While and Repeat-Until**, similar to the last construct but both iterate until/while a condition becomes false or true (in the same manner as defined in most of the programming languages).

---

**Simple Processes.** Simple processes are used as elements of abstraction; a simple process may be used either to provide a view of (a specialized way of using) some atomic process, or a simplified representation of some composite process (for purposes of planning and reasoning) (OWL-S, 2004). Figure 2-4 shows how these relationships among processes are reflected in the OWL-S ontology. Simple processes are not invocable and are not associated with any grounding, but like atomic processes, they are single-step executions.

---

### 2.1.1.3 Service Grounding

The grounding of a service specifies the details of how to access the service. It specifies the protocol and message formats, serialization, transport, and addressing. OWL-S grounding relates concrete and transmittable messages with the abstract inputs and outputs of an atomic process. OWL-S currently maps solely to WSDL descriptions with SOAP bindings.
2.1.2 WSMO

WSMO (Web Service Modelling Ontology) (WSMO, 2004) is an initiative to create an ontology for describing various aspects related to semantic Web services, which development was started by the Semantic Web Services working group of the SDK cluster, which includes more than 50 academic and industrial partners. Its aim was to become a worldwide standard for the semantic description of Web services, providing a machine-interpretable model for describing all the aspects of a semantic Web services. Its main principles, rooted in the those set by the WSMF (Web Service Modelling Framework) (Fensel and Bussler, 2002), are a strong decoupling of the semantic Web services elements; a strong centralized mediation between this elements; and ontological role separation, descriptions of elements are contextualized (e.g. the descriptions that a client handle may be different from those that the service’s hosting system). With these principles WSMO aimed at solving the IT systems integration problem.

In the rest of this section we describe each of the main elements of the WSMO ontology, in a similar fashion that we did for those of OWL-S (section 2.1.2.1 WSMO Ontology Main Elements). Then we will outline the characteristics of the WSML language, which is the implementation language created to represent this service ontology (section 2.1.2.2 WSML).

2.1.2.1 WSMO Ontology Main Elements

Let us describe the main elements that comprise the WSMO ontology. The most important (which relationships are portrayed in Figure 2-5) are WSMO top-level elements, which are the concepts Ontology, Goal, Web Service and Mediator.

![Figure 2-5 WSMO Ontology Top-Level Elements (adapted from (WSMO, 2004)).](image)

2.1.2.1.1 Ontologies

Ontologies provide the terminology for describing the different elements of a semantic Web service. WSMO ontologies are composed of the non-functional properties, imported ontologies,
used mediators, axioms, concepts, relations, functions and instances. Different ontologies can be interconnected or refined using mediators, as we shown later on.

### 2.1.2.1 Web Services

A **Web service** is a computational entity that is able (by invocation) to achieve a user’s goal (WSMO, 2004). Web services are described by means of non-functional properties, functional properties, and behavioural aspects. More precisely:

- **Non-functional properties.** Non-functional properties are attached to almost all the elements described with WSMO. WSMO does not pose restrictions on how non-functional properties are encoded into logical expressions or which is set of non-functional properties that should be considered for describing an element. WSMO, to ease interoperability with other environments, proposes the following terms: accuracy, financial, network related properties, QoS, performance, reliability, robustness, scalability, security, transactional, and trust.

- **Functional properties.** They are defined relating the Web service with a capability. The capability of a service is conformed by a set of axioms that express the service competence. These axioms are the precondition, which specifies the information space of the Web service before its execution; assumption, which depicts the state of the world before the execution of the Web service; postcondition, which describes the information space of the Web service after its execution; and finally effects, a description of the state of the world after the execution.

- **Behavioural aspect.** Associating the service with one or more interfaces sets the behavioural aspect of a service. An interface describes how to fulfil the capability of a Web service.
service. Interfaces (as shown in Figure 2-6) represent a twofold view on the operational competence of the Web service, its choreography and its orchestration. We describe them later on.

2.1.2.1.3 Goals

Goals are representations of an objective for which fulfilments is sought through the execution of a Web service. More precisely, a Goal concept represents the set of services that would potentially satisfy the user desires. Goals are thus descriptions from the point of view of the user, and therefore they realize the ontological role separation of services descriptions.

As we show in Figure 2-7, the main components of goals description structure are very similar to the description of the Web Service concept (as shown in Figure 2-6). They are composed by the same concepts, but now instead of actual services features, they specify desired functional properties and behavioural aspects. As such, as described in Figure 2-7, the relationships between the Goal and Capability/Interface concepts are requestsCapability and requestsInterface. To sum up, a goal is a description of the desired functionality and interface described from the requestor standpoint, which is realized by the system by means of different types of mediators and service and goals orchestration.

2.1.2.1.4 Mediators

A mediator is a component that connects heterogeneous components and resolves mismatches between them (WSMO D29, 2005). Its objective is to overcome structural, semantic or conceptual mismatches that may appear among the components of a WSMO description.
A mediator can have one or several source components and one or several target components. Depending on the types of the sources/targets of the mediator, four types of mediators are considered.

- **ggMediators.** The purpose of ggMediators is to connect goals, providing additional information on their relationships. They are used to achieve goal specification by refinement; goal adjustment by strengthening and weakening; and goal ontologies for efficient management of WSMO Elements. The mediation technique used by ggMediators is called \( \Delta \)-relations.

- **ooMediators.** They are used to carry out ontology integration (i.e. mapping, merging, and alignment of ontologies) as well as for performing the lifting and lowering from the syntactical to the ontological level. They solve interoperability problems between the source ontology of the mediator and the target WSMO top-level element. Thus, their source elements are ontologies (or ooMediators) and their target components can be any WSMO top-level element.

- **wgMediators.** These mediators link Web services with goals, stating that a given Web service (totally or partially) fulfils the goal to which it is linked. wgMediators explicitly state the difference between the two entities. As we portrayed, a goal is a request from the requestor standpoint, this type of mediators are the responsible of bridging between the request and the actual functionality provided by a service.

- **wwMediators.** wwMediators solve heterogeneity problems among Web Services. These discrepancies may occur when a Web Service needs to invoke multiple Web services in order to achieve its capability. A wwMediator can be used to operate between the choreographies of the invoking and invoked Web services.

### 2.1.2.1.5 Interfaces

*Interfaces* describe how the capability of the Web service can be fulfilled (WSMO, 2004), that is which are the steps that the requestor and the service take to achieve the functionality contained in the service capability. Interfaces should specify both the communication and cooperation aspects of the services. Communication as it describes how to communicate with the service such that the service will provide its capability; and collaboration, since it states how the service collaborates with other WSMO services to achieve its capability. These descriptions are referred as service choreography and the service orchestration respectively.
o **Choreography.** It is a decomposition of the capability in terms of interaction with the Web service. It describes how to communicate with the service so that it can provide its intended functionality.

o **Orchestration.** It is a decomposition of the capability in terms of functionality required from other Web services, how to make them cooperate to achieve the overall functionality.

Choreographies and orchestrations both follow a state-based description approach, and are represented directly using abstract state machines (Gurevich, 1995). According to the election of abstract state machines as the underlying model of choreography and orchestration is the minimal ontological commitment; their ability to model any aspect of computation; and their formality. We will discuss thoroughly in 8.3 Appendix III: Abstract State Machines.

### 2.1.2.1.6 Annotations

Though they are not one of the top-level WSMO elements, all WSMO elements presented so far are related with an annotation element. *Annotation* elements provide additional meta-information about a WSMO element, information about its authorship, version control data, human-oriented format specification, etc. As in the case of non-functional properties (see the Web Service description), WSMO in order to be as flexible as possible just contains a recommendation of the set of properties that an annotation should posses. The recommended set is the Dublin Core Vocabulary (Weibel et al., 1998), which among others includes the creator of the description element, any possible contributor, the coverage of the element in textual form, the date of the creation, the publisher etc. WSMO also recommends the use of FOAF vocabulary (Brickley and Miller, 2007) when specifying annotation properties related with people.

### 2.1.2.2 WSML

**WSML** (*Web Service Modelling Language*) (WSML, 2005) is the language (with different variants) created to formalize the afore-described WSMO ontology and all the necessary domain ontologies. The motivation for the creation of WSML is that according to (de Bruijn et al., 2005) the OWL-DL language (used for implementing the OWL-S ontology, as we have already stated) has limitations, both on a conceptual level and with respect to some of its formal properties. To overcome such limitations, several language variants are proposed. These different WSML variants diverge with respect to their level of logical expressiveness; and with respect to the different languages paradigms that they use. All of them are layered, both
syntactically and semantically, and all of them share the same keywords, similar to those of the elements of the WSMO conceptual model.

As we depict in Figure 2-8 there are five WSML variants, namely WSML-Core, which intersects Description Logic and Horn Logic and provides the basis for all the other variants; WSML-DL, an extension of WSML-Core in the direction of Description Logic; WSML-Flight and WSML-Rule logic programming counterparts of WSML-Core; and finally WSML-Full, which unites the functionality of all the enumerated WSML variants.

2.1.3 IRS Framework

IRS (the Internet Reasoning Service) Framework (Motta et al., 2003)(Domingue et al., 2008a) is a semantic Web services framework, which includes a service description ontology and the necessary infrastructure to execute such semantically annotated Web services. It was developed by the Knowledge Media Institute (KMI) of the of United Kingdom Open University. Its last two incarnations are IRS-II and IRS-III and the main characteristics of their service description ontologies are:

- **IRS-II Framework** (Motta et al., 2003) service ontology elements are almost equal to those proposed on the UPML (Unified Problem-solving Method description Language) framework (Fensel et al., 2003). Therefore, the descriptions that it handles are similar to the Problem-Solving Methods Model that we depict thoroughly in section 4.2 Problem-Solving Methods Model. In the shake of completeness, this model is composed by domain models, as descriptions of the domain of an application; task models which provide a generic description of the operations to be solved; Problem-Solving Methods models that provide abstract implementation and independent descriptions of reasoning processes applied to solve tasks in a specific domain; and bridges, that specify mappings between the different model components within an application. We represent these elements in Figure 2-9.
These elements are represented in OCML (Motta, 1998), an Ontolingua (Farquhar et al., 1997) derived language (implemented with Lisp plus some extensions). OCML has enough expressiveness to specify task specifications and service competencies, and also provides operational support to reason about these elements.

- **IRS-III Framework** (Domingue et al., 2008a) service ontology is an extension of IRS-II service ontology to make it WSMO compliant. Its elements has been adapted and complemented to reflect the WSMO knowledge model presented in the previous section.

IRS-III service ontology is the last incarnation of the IRS Framework service ontology and therefore we restrict ourselves to its description. Taking into account that its core service ontology is WSMO, its description is mostly identical to the one contained in section 2.1.2 WSMO; we will just outline their main differences that were detailed in (Domingue et al., 2008a) and which among the most important are:

- **Explicit inputs and outputs definition.** IRS-III includes the explicit specification of inputs and outputs roles in the definition of services and goals. Goals and services descriptions in WSMO are more abstract than IRS-III ones; whilst IRS-III descriptions are more effective in cases where the described goal or service is simple.

- **Modelling made at concept level, not only at the instance level.** IRS-III allows reifying the main WSMO elements, enabling for example the creation of taxonomies of services and goals. WSMO descriptions are just composed by instances of the top WSMO description elements.

- **Implementation language.** IRS-III implements the WSMO ontology using the same representation language used to implement the IRS-II service ontology (i.e. OCML).
2.1.4 Limitations

The presented frameworks were created for the description of semantic Web services. The question before using them for describing semantic Grid services should be whether Grid services and Web services are identical, or at least Grid services are a subset of Web services. Otherwise it would be a mistake to use a formal framework designed for simpler entities. The problem is that this is the case with Web services and Grid services, and we have identified a list key of necessary extension points over the state of the art frameworks to achieve a sound Grid services description framework (we published them in (Goble et al., 2005b)). They are:

- **Non-Functional Properties.** Many concerns and activities such as trust, quality of service, workload distribution, services discovery and composition are non-functional properties dependant. Among the described semantic Web services OWL-S and IRS focus mainly on functional properties currently, non-functional properties have a secondary role.

- **Virtual Organizations description.** This is a crucial concept in Grid computing that is not present in the Web. Grid is about sharing resources under a certain set of rules; resources that are virtualized by services. The description framework provides a formal and explicit description of the institution that is created by the sum of these services; the rules that govern the interaction between the entities involved; and the entities themselves (i.e. providers, consumers, and all the other roles that may coexist in a Virtual Organization). Note that the concept of Virtual Organization does not exist in the Semantic Web services field since Semantic Web services are considered as isolated elements (as Web services are).

- **Provenance.** Provenance information gives the origin and metadata information of a concrete enactment of a Grid service. It also provides and allows us to characterize what caused things, how "things" depended on others and resulted in specific states (Moureau et al., 2010). With this information we are able to interpret the process enactment results. Provenance turns out to be very important in Grid environments, since Grid applications often deal with long time running experiments. In such scenarios, knowing which data and services were used to generate the results, who started and who was involved the experiment, and more importantly, the reasons of possible failures, is crucial. None of the described semantic Web frameworks take into account provenance description.

- **Complex interactions.** Initially, Web services interactions (and as a consequence also for Semantic Web services) were usually composed by the pair invoker/invoked service. This scenario can be considered as a refurbished XML version of the classic client/server
interaction (see (Alonso et al., 2003)). Even though there are Web services initiatives like (WSCI, 2002) (WSCDL, 2005) that propose complex interaction models, the mainstream of Web services are still bounded to simple interactions models. Semantic Web services initiatives, as were created to annotate the existing services, also adopted this simple model of interaction, there is no way to describe Grid environments complex interactions, which are those that involve:

- **Multiple parties and multiple stages.** Complex interactions may engage multiple (i.e. more than two) cooperating participants. Each of these participants should be characterized in a formal way.

- **Multiple views.** The description of complex interaction implies the use of a multilayered approach. They should range from a global perspective to the local perspective focused in the view of each concrete participant. This description may involve the definition of the different roles of the entities that participate in the global model.

- **Complex message exchanges.** The interactions between two services can be more intricate than a request/response pattern.

  - **Resources.** Grid services virtualize real world computer resources; the intention of virtualization is to abstract them. In semantic Web services description frameworks, the concept of the real world and the concept of the domain model (the abstract and formal representation of the world) are often confusing, and therefore confused.

  - **Transformational systems.** Currently Web services, and consequently Semantic Web, services are considered transformational systems. It makes perfect sense in the case of Web services, since they were considered stateless interfaces of a set of operations. The interaction with such systems consists just in the provision the inputs and the recollection of outputs. After these activities are carried out, the service finishes. Grid services exhibit a different behaviour that involves long-lived conversations and complex external behaviours and cannot be considered transformational systems.

  - **Grid services instances.** This is one of the more subtle differences between Semantic Web services and Semantic Grid services. The discovery, creation and invocation of transient a Semantic Grid service’s instances are a necessity. The possibility of specifying a concrete Semantic Grid service instance or even a scheduled invocation of one of it operations is not contemplated by current Service Web service orchestrations and descriptions. Because of the service statelessness, the presented semantic Web services work at a “class of services” level. We need a framework that allows working at an “instance of service” level instead,
since they may virtualize stateful resource, and none of the above-described description framework considers the lifecycle management and specification of a concrete instance of a Grid Service.

Up to now we have identified a set of limitations that make current semantic Web services descriptions proposals not suitable for describing semantic Grid services. Therefore, these frameworks should not be used directly; they must be extended to cope with all these Grid specific requirements. But why don't we just pick one of the aforementioned semantic Web services initiatives and adapt it to cope with the new Grid requirements? The problem is that we identify a more profound drawback in all of them. They all work at the symbol level and it is clear that higher-level descriptions would bring many benefits. Designs made at the symbol level are less reusable and shareable and handling directly representational languages is more prone to inconsistencies and errors. But more importantly, the quality of the design is leveraged due the possible limitations and constraints imposed by the target low-level language. All these downsides are a direct consequence of broking the separation between the design and implementation phases and should be avoided.

To sum-up we have graphically situated OWL-S, IRS (both IRS-II and IRS-III) service ontologies and WSMO with respect to their complexity and with respect to their level of abstraction (see Figure 2-10). The abscissa represents the complexity of the elements that the description framework is able to specify; the ordinate represents the level at which the design is made.

Figure 2-10 Representation of different description frameworks regarding complexity and design level.

OWL-S is completely formalized using the OWL language, which is one major drawback. Description logics are very useful for the definition of classes of individuals with certain common static characteristics (i.e. terminological knowledge); but they are not well suited to describe the operational semantics of processes such as the orchestration or the choreography of
a service. Therefore OWL-S becomes almost only a valid option for the description of taxonomies of services and the annotation of inputs and outputs.

IRS-II service ontology is closer to our goal of achieving a high-level design of services. Nevertheless, this approach might be valid just if we consider Web services almost as atomic function invocations, since the solely use of Problem-Solving methods for describing services restricts too much the entities that this framework is able to describe. In consequence, they are not suitable for describing services that imply complex interactions. IRS-III service ontology converges with WSMO, providing a more complex and complete framework. Nevertheless, it lowers the level of its descriptions and keeps using OCML, what unavoidably further restricts their designs.

Finally, WSMO is a description framework that achieves its main objective, the creation of a framework for describing various aspects related to semantic Web services. WSMO provides complete descriptions for services, including both the annotation of the elements handled by services as well as the execution semantics of these services. WSMO also takes into account the mediation role of Web services, becoming mediators one of its key defining elements. The problem with WSMO is that was created for describing Web services, and therefore it does not include concepts such as Virtual Organizations, resource sharing policies, interactions that specify the role that several participants play, etc. Regarding the abstraction level, WSMO it is too attached to the chosen description mechanisms. Just as an example, orchestrations and choreographies are described directly using Abstract State Machines (see section 8.3 Appendix III: Abstract State Machines for a description of them), which is a perfect approach for formalizing the operational semantics of processes, but it is not suitable for designing them at all.

2.2 SEMANTIC WEB SERVICES PROVISIONING ARCHITECTURES

In this section we are going to provide a description of the two main architectures for the provisioning of semantic Web services, which are IRS Framework (2.2.1 IRS ) and WSMX (2.2.2 WSMX). After their descriptions we will discuss the limitations that we identify in these proposals.

2.2.1 IRS Framework

As we have already stated, the IRS Framework also includes an infrastructure to provision services described with the IRS Framework service ontology. We describe the architecture of its last incarnation (described in (Domingue et al., 2008a)), which is the one that provisions
services described using the IRS-III service ontology. Previous versions of the architecture, from an architectural standpoint and despite the differences on the knowledge model that they handle, are basically identical. We represent it in Figure 2-11, and in the following sections we describe each of its main components.

![Figure 2-11 IRS Framework software architecture (adapted from (Motta et al., 2003)).](image)

The IRS Framework architecture is classical client-server architecture; it is designed as a broker for service provisioning. The only noticeable change to the old paradigm is the inclusion of what is referred as publishing platforms. Therefore, the IRS Framework architecture main components are the IRS server, the IRS client, and the publishing platforms. In the following sections we describe them, and their most important components.

### 2.2.1.1 IRS Server

The IRS Server is contained inside an HTTP server, as it is the transport protocol used to send and receive the SOAP messages that the server consumes. Its main components are:

- **Semantic Web services library.** The service library is a knowledge base that stores all the semantic descriptions for each of the services, goals, adapters, domain models, etc.. All these descriptions are implemented in OCML (Motta, 1998).

- **Choreography interpreter.** This component interprets the formal descriptions of the IRS services choreographies. The choreography of a service is the number of communication messages interchanged to achieve the goal associated with the service, which translates into the performing of a number of discrete steps which execution depend on the current state of the interchange. IRS, as WSMO, uses abstract state machines (Börger and Stärk, 2003) to
represent these interactions with the client. As described in detail in (Domingue et al., 2006), choreographies are thus divided in:

• **The set of rules of the abstract state machine.** These are the transition rules that define which represent the interaction between IRS-III and the Web Service on behalf of an IRS Client. They are applied when executing the choreography. This model is executed at a semantic level when IRS-III receives a request to achieve a goal.

• **The grounding of each of the operations involved in the choreography.** That includes the conceptual representation of the operations involved in the invocation of a Web Service; and the mapping of the conceptual inputs/outputs with the concrete SOAP binding of the Web service.

  o **Orchestration interpreter.** This component realizes the orchestration of complex services, described in terms of simpler ones. This complex Web services functionality is expressed in terms of sub-goals that represent what the composing services should achieve. The concrete services that are invoked are selected at run-time, in terms of the concrete execution circumstances and the user request.

• **The orchestration is goal oriented.** The basic unit is the goal invocation. Goals are orchestrated using a set of control and data constructs. These primitives include *Orch-sequence*, that specifies a set of goals to be invoked sequentially; *Orch-if*, that allows the definition of several branches of execution depending on a Boolean condition; *Orch-repeat*, allows the definition of a conditional loop; *Orch-get-goal-value*, represents the result of the last invocation of the declared goal; and *Orch-return*, that finalizes the orchestration returning a value. The underneath representation mechanism for orchestration built with this primitives are also abstract state machines, so all of them will be also formalized using transition rules as was the case of the choreography.

• **The orchestration includes interleaved meditation.** The resolution of mismatches between data and goals is included in the orchestrations, supported by means of mediators and the mediation handler.

  o **Mediation handler.** This module interprets the different types of mediators that WSMO defines (see 2.1.2.1.4 Mediators). Specialized mediators are described declaratively, providing conceptual mismatches or specifying the mediation service where necessary. The mediation handler interprets each type of mediator accordingly during selection, invocation and orchestration.
Work Objectives

- **Invoker.** The invoker is the component that communicates with the publishing platforms acting like a communication medium that gathers all the necessary inputs, and then delivers the results either to the client or the correspondent orchestrated service.

### 2.2.1.2 Publishing platforms

*Publishing platforms* are the link between the deployed Web services and their description hold in the semantic Web services library. IRS-III provides publishing platforms to support grounding to stand-alone Java and Lisp code, and WSDL described Web services.

### 2.2.1.3 IRS Client

*IRS Client* enables the capability driven invocation of Web services. The external agent that wishes to invoke a service needs to specify the goal of its invocation (in IRS-II was the task it wanted to perform) plus its inputs. From this information the IRS broker is able to determine the goal to be solved; it locates appropriate set of Web service semantic descriptions by means of wg-mediators; and then invokes the correspondent Web services by means of the publishing platforms.

### 2.2.2 WSMX

*WSMX (Web Service Modelling Execution Environment)* (WSMXa, 2005) is an environment for the execution, dynamic discovery, mediation and invocation of Web services. As in the case of IRS, the description model of WSMX is WSMO. In fact, WSMX is the reference architecture for WSMO services provisioning.

Services described using the WSMO ontology can be stored in the WSMX environment. The description of these services is implemented in WSML (not as the case of the IRS Framework that were implemented in OCML). As in the case of IRS, external entities are not aware of the services registered in the environment, neither their capabilities. The client just asks the environment to achieve a goal, and the system is in charge of resolving any mismatch between the desired goal and those that the registry contains. Then the system must satisfy the user goal, searching by means of mediators the service that is better to carry out the desired goal. Therefore in this process, WSMX makes a heavy use of mediators in order to solve any mismatch that might appear between the vocabulary of the invoker and the ontologies that the goals and services in the WSMX repository might use; and also mediators that solve discrepancies between the invocation processes carried out by the external agent and the actual invocation choreography of the service. Thanks to the machine-interpretable description of
services all these activities are performed transparently and automatically without the user intervention.

One of the great features of WSMX is that apart from being a service provisioning architecture, it was conceived as a service-oriented architecture. As such, and as we shall describe in section 5.1.2 Service-Oriented Paradigm Principles, service-oriented architectures exhibit good software design characteristics. WSMX constituent services are low coupled between them, they possess a well-defined and self-contained functionality. Consequently, the overall functionality that this platform should provide has been well identified and grouped in a set of nodes that perform orthogonal activities. The easiness for replacing such nodes is granted by the low coupling among these nodes. But moreover, as described in (WSMXa, 2005) these components do not necessarily execute in the same address space; they may very well execute on different machines in different network locations. This enhanced the ability of WSMX infrastructure to scale and recovery from failures.

Let us focus on the description of each of the entities and modules that shape the WSMX architecture (WSMXb, 2005). As we represent in Figure 2-12 around a WSMX environment service requesters and service providers are situated. A service requester becomes any external user, agent acting on behalf on an user, or back-end application that interacts with the architecture in order achieve a goal; whereas service providers are the external entities that supply the services that might achieve these goals, and which descriptions are stored in the WSMX environment. Regarding WSMX modules, we must stress that WSMX is meant to be a reference implementation of WSMO. As such, it provides an implementation of all the modules that appear in Figure 2-12, but in a real-world usage scenario it is not mandatory to deploy all of them. Administrators can decide which of them are necessary, or even they might define and use their own modules. Let us describe the most important WSMX modules:

- **WSMX Manager Core.** This component is the backbone of the WSMX infrastructure, the rest of the components are registered on it, managed by it, and communicate through it. This design choice is important, since all the interactions between components are coordinated and pass through it, situation that might result in a bottleneck. If you observe, in Figure 2-12 all the other WSMX modules are connected to this module by means of a wrapper and an interface. All of them share the same structure, a core functionality that corresponds with each of the basic activities that WSMX provides, plus a surrounding wrapper. These wrappers are obtained automatically from the well-defined interfaces that WSMX modules expose, and are used by the WSMX manager core to interact with them.
Figure 2-12 WSMX architecture (adapted from (WSMXb, 2005)).

- **Discovery Component.** This component realizes the service discovery activity, or what is the same, the mechanisms for finding services described in WSMO that posses certain characteristics (i.e. as we mentioned earlier in this chapter, services that adhere to some client-specified constraints). WSMO contemplates three levels of discovery (WSMO D5, 2004), keyword-based discovery, discovery based on simple semantic descriptions (i.e. using concepts and taxonomies), and discovery based on rich semantic descriptions of services (i.e. use of axioms, non-taxonomic relationships, etc.).

- **Resource Manager.** The resource manager is a module that acts as an interface that virtualizes data access, unifying the handling of information of different nature. This information includes WSMO descriptions implemented in WSML (services, goals, mediators, etc.), services WSDL documents, etc.

- **Communication Manager.** The communication manager encapsulates and isolates the logic related with the underneath communication infrastructure. It accepts the incoming and outgoing messages, deals with the different transport technologies, security issues, communication protocols, etc. None of the end points of the communications made in WSMX are aware of each other. Instead they communicate through the communication manager, which acts as the receiver from the invoker side (activity realized by the receiver...
submodule); and the invoker from the invoked service perspective (activity carried out by the Invoker submodule). Finally, the communication manager handles all correspondence between the abstract WSMO service endpoint and its concrete implementation (a WSDL Web service) by means of the grounding submodule (we represent the submodules in Figure 2-12, below the communication manager module)

- **Selectors.** The selector is a component that given a candidate set of discovered services, it chooses which one is the more suitable for being invoked. Note that determining which is the best candidate is heavily influenced by the chosen criteria or heuristic.

- **Mediators.** As we have already stated, mediation is a key activity in the WSMO framework, and therefore it should also be in WSMX. WSMX implements two types of mediation depending on the discrepancies to be solved. Consequently there are two different types of mediators, namely data mediators and process mediators. More precisely (WSMXa, 2005):
  - **Data mediators.** A WSMX data mediator component has the role of reconciling the data heterogeneity problems that may appear during discovery, composition, selection or invocation of Web services. The knowledge necessary to perform this activity is obtained from the mappings made at the ontological level contained in mediators definitions (see 2.1.2.1.4 Mediators). These mappings can be used as rules for translating from one vocabulary to another, and this is precisely what this component does.
  - **Process Mediator.** A WSMX process Mediator has the role of reconciling the public process heterogeneity that can appear during the invocation of Web services. The knowledge necessary to perform this activity is obtained of the machine-interpretable description of the process necessary to invoke the service, i.e. its WSMO choreography. Analyzing both the choreography and the message exchange performed by the invoker, this components identifies different discrepancy situations among them, and solves them invoking to different mediation services.

- **Reasoner.** The reasoner module provides reasoning services for the rest of the modules of the WSMX infrastructure. Reasoning is involved in many of the activities that other modules perform, such as for the mapping process of data mediation, the discovery of services that match the requester's goal, validation of a complex service orchestration, etc.
2.2.3 Limitations

Regarding the presented semantic Web services provisioning architectures we identify the following limitations:

- **Centralized Client/Server paradigm.** Both IRS Framework and WSMX share the same architecture paradigm. Both are client/server architectures that rely on monolithic servers, especially in the case of IRS Framework that can be considered as an archetype of such type of architecture. WSMX uses JMX technology that allows the definition of this architecture as a collection of easily interchangeable plug-ins. Nevertheless, it does not make it a distributed architecture but a modularized server. This architectural choice brings many problems, among others a decrease in the system scalability, restricting the throughput of the overall system to the computational capacity of a single node, and finally a decrease of the system reliability.

- **Static conventional architectures.** Both are conventional architectures with no dynamic adaptation mechanism. There is a total absence dynamic features such as additional resource allocation in order to deal for example with peaks of requests traffic; active monitoring an automatic configuration processes to update the environment optimally in cases such as the deployment of new components or removal of existing ones at runtime; self-healing, to deal with fatal errors, etc.

  ![Interaction modes table](image)

  *Figure 2-13 Interaction modes table (adapted from Mühl et al., 2006)).*

- **Request/reply mode of interaction.** Both IRS Framework and WSMX only consider the request/reply model of interaction. We represent a table with of possible models of interaction in Figure 2-13. The different types of interaction modes contained in the table are defined in terms of which is the initiator, that is whether the consumer or provider is the entity that begins the interaction; and its addressing mode, that distinguishes whether the interaction is addressed directly to a particular entity or it remains indirect and thus unknown. With the exception the request/reply the rest of interaction modes are
unsupported, ignoring thus its associated benefits. This limitation is especially important in the case of the callback interaction model, which would bring, if considered, the possibility of providing a semantic-aware subscription-based system; and in the case of event-based interaction model, which benefits if considered are well studied and we have already outlined in section 1.4 An Event-Driven Architecture for the Semantic Grid and the Semantic Web.

### 2.3 DISTRIBUTED EVENT-DRIVEN ARCHITECTURES

In this section we provide an overview of the related work regarding event-driven architectures. We focus only on those event-driven systems that provide a distributed notification topology, as we are interested only in truly distributed systems. In (Tarkoma and Raatikainen, 2006) a complete taxonomy of the main event-based systems is provided, in Figure 2-14 we represent those event-based systems that are distributed. From each of these approaches we offer a brief description of its main characteristics and features, focusing on summarizing their approach to notification representation and routing. After their descriptions we will discuss the limitations that we identify in these proposals, once again centring our attention on the scope of the presented work.

![Figure 2-14 Taxonomy of distributed event-based systems (adapted from (Tarkoma and Raatikainen, 2006)).](image)
2.3.1 Scribe

Scribe (Castro et al., 2002) is a topic-based publish-subscribe, large-scale, and fully decentralized application-level multicast infrastructure. Initially was developed by Microsoft Research.

Scribe is built on top of Pastry (Rowstron and Druschel, 2001), an implementation of a distributed hash table (DHT) proven to be a scalable, self-organizing peer-to-peer location and routing system. As depicted in (Castro et al., 2002), Scribe system consists of a network of Pastry nodes; each node runs the Scribe application software. The Scribe software on each node provides the forward and delivers methods, which are invoked by Pastry whenever a Scribe message arrives. Notification forwarding, it is made using group restricted multicasting. Scribe nodes may create a group and become and become the rendez-vous point for that group, which means that it is able deliver messages to all members of the group using a multicast best-effort delivery. Subscription description thus is expressed in terms of group membership. If a subscriber is interested in certain notification topic it must join the appropriate group where notification of this topic are being multicasted.

2.3.2 Elvin

Elvin (Segall et al., 2000) is an event system developed by the Distributed Systems Technology Centre (DSTC) in Australia.

Elvin is a general-purpose event notification service designed as a distributed client/server architecture. Consumers announce their subscription descriptions to a central notification server, a proxy that provides a single-subscription address space. Under the hood, Elvin provides support for the clustering, federation and error recovery of notification servers in order to support this single point of access logical view in a scalable fashion.

An event notification is represented as a set of tuples, each of which containing the notification attribute name and the attribute value. Subscription expressions are defined using predicate-based expressions (i.e. attribute, operator, value). The routing of notifications is based on content-based filters, but unlike the other event systems, expressions are evaluated using Lukasiewicz's tristate logic (i.e. the result of the evaluation of a filter can be either true, false, or indefinite). Subscription description messages, instead, are delivered using a multicasting approach.

Elvin team also has carried a great effort to reach the wider use possible. On the one hand, the Elvin subscription client has been ported and has bindings to many languages and
technologies (Java, C, C++, and long etcetera); and on the other hand, the development of the Elvin protocol in the Internet Engineering Task Force (IETF).

2.3.3 JEDI

JEDI (Cugola et al., 2001), the Java Event-Based Distributed Infrastructure, is a distributed event system developed by Cefriel at Politecnico de Milano. It was developed initially to implement the Orchestra Process Support System (OPSS) workflow management system (WFMS). As such, it was not designed to be deployed as a widespread distributed system.

Event dispatchers and active objects compose the JEDI architecture. Event dispatchers correspond with event notification routers, whereas active objects are the consumers/producers of event notifications. The topology of the system is configured as a tree, which nodes are the dispatchers and the leaves are active objects. Event subscription/unsubscription requests and event notifications are propagated by each of the event dispatchers upwards towards the root dispatcher. Once a notification is delivered to an event dispatcher, it checks the subscriptions of all its descendants in the tree to decide whether to deliver the notification to them or not. This approach implies that on the one hand each node must have complete knowledge about the subscriptions of all the nodes beneath it in the topology tree; and on the other, it also entails that all the topology configuration messages and event notifications pass through the root node, which might become a bottleneck.

Event notifications in JEDI are represented as tuples composed by each notification property and the list of its associated values (all of them strings). Consumer subscriptions are templates that represent regular expressions, and the filtering of notifications is performed using string matching.

From its initial implementation JEDI has been extended to solve its initial fault tolerance and scalability issues in order that it can be effectively used as a distributed communication infrastructure. It has also been extended to support for mobile clients (Cugola and Jacobsen, 2002), allowing active objects to move from being connected to one event dispatcher to other; and to support dynamic reconfiguration of the system topology (Picco et al., 2003).

2.3.4 SIENA

SIENA (Scalable Internet Event Notification Service) (Carzaniga et al., 2001) is a multi-broker event notification service that targets ant Internet-scale deployment. It has been developed at the University of Colorado.
SIENA is very flexible with regard to the shape of the system topology, supporting three topology arrangements, namely an acyclic peer-to-peer topology, a hierarchical tree-alike topology and a generic peer-to-peer topology. The notification brokers use a cover-based algorithm that allows brokers to only be aware of the subscription/advertisement descriptions of its directly connected neighbourhood. The propagation of notifications and descriptions is made quite differently. In the case of event notifications, it is made in terms of completely local decision, a notification is delivered to a neighbourhood just applying the filters that describe each neighbourhood. In the case of subscription descriptions, SIENA proposes a shortest-path delivery heuristic that results in the need for a wider knowledge about the overall topology, but a more efficient propagation of the descriptions.

An event notification in SIENA is represented as a set of typed attributes; becoming the notification a set of triples attribute type, attribute name, and attribute value. There is no abstract representation of notifications; notification types or hierarchies of notifications cannot be defined. Subscription/advertisement descriptions are formalized as the conjunction of a set of Boolean filtering clauses. Each filtering clause is formed by a triple that includes the name of the notification property, an operator, and a value.

### 2.3.5 REBECA

REBECA notification service (Fiege and Mühl, 2000) is a project carried out at the Technical University Darmstadt.

The notification routing topology of REBECA is shaped as an acyclic graph, composed of three types of brokers, namely local brokers, those that are directly connected with producers/consumers; border brokers, which connect the local brokers with the communication middleware; and finally, inner brokers, that do not connect directly with producers or consumers, and represent the backbone of the brokering infrastructure.

Event notifications in REBECA are modelled as a set of triples that represent the name and value of each of the notification properties plus its type. Contrary to other approaches that offer a finite set of types, a mechanism for adding new types is provided. Regarding the delivery of notifications, REBECA supports various types of content-based algorithms such as flooding, simple, identify-based, covering-based, and merged-base (see Mühl and Fiege, 2001). Furthermore, REBECA proposes the concept of scopes, which are sets of producers and consumers and are defined to limit the visibility of notifications of the enclosed components.
2.3.6 Gryphon

Gryphon (Storn et al., 1998) was developed at the IBM Distributed Messaging Systems group at T.J. Watson Research Center. Gryphon is a Java-based publish-subscribe message broker and is now part of the IBM’s WebSphere suite, and can be considered a low-level middleware event notification system that have features such as guaranteed delivery, durable subscriptions, recovery from server failures and security.

Gryphon models interaction in an event-driven system in a completely different way to the other initiatives presented in this section. Gryphon introduces the notion of Information Flow Graphs (IFG) (Banavar et al., 1999), an information flow model for messaging. IFGs specify the exchange of information between information producers and consumers. This exchange of information is an abstraction from the underneath physical system, and can contain both stateful event interpretation functions that associate even notifications with the happening of interest that they signify (e.g. alarms, trends, etc.); and stateless event transforms, which are used to combine events from various sources. From this logical view the system automatically maps to the underlying physical topology. Gryphon physical topology of event brokers is completely static, configured at design-time and contained in a configuration file.

Regarding notification delivery, Gryphon uses a distributed content-based approach similar to the one provided by SIENA. But unlike SIENA, in the case of subscription description Gryphon propagates consumer descriptions to all the brokers in the system.

2.3.7 CORBA ES and CORBA NS

Finally, we briefly describe two specifications, CORBA ES (CORBA Event Service) (OMG, 2000) and CORBA NS (CORBA Notification Service) (OMG, 1999). CORBA ES defines a communication model that allows an object to accept registrations and send events to a number of receiver objects. CORBA ES was initially proposed to complement the standard CORBA client-serve communication model (i.e. request/reply interaction model in Figure 2-13). CORBA NS extends the functionality and interfaces of CORBA ES to support better interoperability and the inclusion of event filtering.

The CORBA ES provides two models for initiating the transfer of events between suppliers and consumers, the push model, in which producer send events to consumer by means of channels; and the pull model, the consumers actively request events from the suppliers. In either case, producers publish their notification on a given set of the set of shared channels, and consumers obtain their desired notifications by listening to a set of these event channels.
Both initiatives possess a standard data structure for event notifications that consists of a header and a body. The type and name of the event are defined in the header. The body of the notification contains the properties of the notification (i.e. those used for filtering notifications) and the payload data. Additionally, CORBA NS provides mechanisms for filtering event notifications, allowing the definition of constraints that can be used to match the event notification type or property values.

2.3.8 Limitations

Let us describe each of the different routing techniques that are used by the described approaches following as guideline the one proposed in (Tarkoma and Raatikainen, 2006) and (Mühl et al., 2006), and for each discuss which are its main characteristics and which are its negative aspects and limitations.

- **Channel-based routing.** Channel-based routing (or topic-based if channels are named) is used by initiatives such as CORBA ES, CORBA NS and Scribe. Its rationale is that producers and consumers connected to the system are aware of a set of communicating channels. Producers publish their notification on a given set of those shared channels, and subscribers obtain their desired notifications by listening also to a set of these event channels of their choice. This approach is too static; in the sense that new channels need to be added to the system each time a new type of events is introduced. Moreover, channel-based routing introduces a huge coupling between the even-driven system and the descriptions of its consumers and producers, since they have to be described in terms of these channels. And finally, this approach is too coarse-grained; consumers and producers descriptions are reduced to the set of channels they are listening, without the possibility of further refining such preferences.

- **Subject-based algorithms.** Subject-based algorithms, as described in (Oki et al., 1993), use string matching for notification selection. Notification producers label their notifications with a subject, usually following a hierarchical schema similar to file names. On the other side of the interaction, subscribers express their notification preferences using regular expressions, for each notification in the system a template matching is made in order to determine whether the notification should be delivered to a consumer or not. We consider type-based algorithms as a subcase of subject based algorithms, in which there is a lattice that defines a hierarchy of types. The problem with such approaches is that on the one hand they operate at a very low syntactic level, using just string comparison as a similarity
mechanism. On the other hand, they are too simplistic and still too coarse-grained, as they do not consider the whole notification properties to route them, but only a string tag.

- **Content-based algorithms.** Content-based algorithms are take into account the whole notification content in order to decide whether to forward it or not. These algorithms are used by most of the presented event-driven systems (e.g. SIENA, REBECA, Elvin, Gryphon, etc.). In such approaches, each broker possess a routing table that contains pairs \((F_i, D_i)\). \(F_i\) represents a filter, a Boolean function that evaluates whether notifications should be delivered, and \(D_i\) a destination, a directly connected broker, producer or consumer. Each time a broker receives a notification, the broker must determine to which of its surrounding brokers should be delivered. It runs through each entry on the table and evaluates the filter of each destination. This process is what we refer as a brokering step. The overload introduced by each of these brokering steps may affect seriously the overall system scalability, since if it takes too much time in each step the broker might end up overwhelmed by the number of pending notifications. As we illustrate in Figure 2-15, this worsened in the innermost notification brokers where as the size of their routing table grows exponentially.

Figure 2-15 Content-based routing brokering step complexity sources.

In order to make the brokers more diligent in each brokering step we should take into consideration:

- **Number of entries.** As we represent in Figure 2-15 the number of entries is one of the axis of the complexity of each brokering step. Different techniques have been applied to reduce the size of routing tables for content-based algorithms. (Mühl et al., 2006) provides a complete description of these algorithms that exploit similarity, inclusion and merging relationships between entries in order to combine them, reducing thus the number of entries of the table. Nevertheless, in case that the description of event-driven
services that we handle are more complex than those used by current approaches, determining the similarity, inclusion and merging relations would be very difficult (if not impossible). An algorithm capable of creating compact routing data structures in the case of complex advertisements and subscriptions expressions is still an open issue.

- **Filter complexity.** The complexity of the filters, the second axis of complexity that we represent in Figure 2-15, is determined by the time it takes to evaluate each of the filters for a given notification. This complexity is directly related with the length of filter expressions, which are determined by the complexity of the event notifications descriptions. None of the approaches in the state of the art have addressed this issue since they propose rather simple event representations.

On the subject of event notifications representation, all the presented initiatives employ a very low-level notification model. Event notifications are in most cases specified as sets of tuples. Higher-level representations of notifications, and the definition of routing algorithms that take advantage of such representations, have not been proposed.
Chapter 3  Work Objectives

This chapter presents the thesis goals, together with the open research problems that we aim to solve. Besides, we detail the contributions to the current state of the art, the work hypotheses, the assumptions considered as a starting point for this work, and the restrictions of the results presented.

In this chapter we start by presenting the goals that motivate this dissertation. After that, we present the list of open research problems that we have identified; and we detail the contributions to the state of the art that the achievement of the dissertation goals entitles. Finally we enumerate the work hypotheses, assumptions and restrictions that we have considered as a starting point or whilst we carried out the presented work. Most of these topics are presented in two groups for the shake of clarity: one of those related with the knowledge level description framework; and the other with those related with the event-driven architecture.

3.1  GOALS

The goals of the work presented in this document are to provide both:

- A knowledge level description framework rich enough to express the semantics required for describing services (both semantic Web services and semantic Grid services) and Virtual Organizations.

- An event-driven Service-Oriented Architecture for the Semantic Web and Semantic Grid that using the knowledge level description framework, and by means of its self-management capability, is able to achieve high-scale provisioning of semantic services.
Work Objectives

3.2 Open Research Problems

In this section we enumerate the main open research problems that we identified in order to achieve the dissertation goals, both those related with the description framework (section 3.2.1 Knowledge level Description Framework) and with the event-driven architecture (section 3.2.2 The Event-driven Semantic Service-Oriented Architecture).

3.2.1 Knowledge level Description Framework

In order to deliver the first objective (the knowledge level description framework for Grid services and Virtual Organization), the following non-exhaustive list of open research problems must be solved:

- Grid specific framework. All the Semantic Web services technologies and description frameworks (i.e. IRS, OWL-S, WSMO, WSDL-S, etc.) focus on the semantic description of Web services. Semantic Grid services and Semantic Web services initially aim at similar objectives and research problems (e.g., interoperability, complex processes description, etc.); and the underneath technologies are the same (SOAP, REST, WSDL, etc.), but as was discussed in Chapter 2 State of the Art, semantic Web services frameworks can not be adopted as off-the-shelf solutions due the different natures of Grid services and Web services on the one hand; and because none of these initiatives addresses the description of Virtual Organizations on the other.

- High-level description framework. A high-level description framework must be envisaged, as described (NGG3, 2006). Current initiatives handle directly representational languages such as FLogic, OWL, OCML, etc. and therefore a description framework that will allow the creation of high-level reusable designs is still an open issue. This framework should allow:

  - The definition of high-level designs. A high-level design sketches an overall view of how a system should work; and also makes explicit the top-level components that comprise the system (and how they are related).

  - The reusability of the designs. Designs should be reusable in two senses: designs, and parts of them, should be easily used and repurposed across different domains and scenarios; and from a single design we should easily obtain different representations, with different languages, expressivities and granularities.
Work Objectives

- **Annotation methods and techniques.** From a methodological perspective, there is a lack of integrated methods and techniques that support annotation and design of services (both semantic Web services and semantic Grid services) and provide a guideline for the complex endeavour of creating Virtual Organizations descriptions.

### 3.2.2 The Event-driven Semantic Service-Oriented Architecture

The second objective (*the event-driven service-oriented architecture*) unveils also open research problems that should be elucidated.

- **An event-driven architecture.** Current semantic services provisioning architectures are rooted in the reply/response model of interaction. It is an open issue the proposal of a semantic services architecture based on an event-based model of interaction. That includes the definition of components that make use of formal semantics for each of the core activities performed by an event-driven system, which are event brokering, composite event detection and event handling.

- **Peer-to-peer knowledge-based routing.** Both IRS Framework and WSMX propose a centralized approach to service discovery. In order to perform knowledge-based notification routing we need an analogous process, but in a distributed and peer-to-peer fashion. We have found that a similar (but simpler) process is provided by the SWAP architecture. Despite SWAP just aimed at the forwarding of queries, this forwarding was made on an intelligent fashion on the basis of the content present at the knowledge bases that conform its peer-to-peer network. Thus we can generalize SWAP architecture so that it uses several domain ontologies and an event ontology on the one hand; and the description of the expertise of each of the peers must be upgraded to describe generic services.

- **Filtering scalability vs event safety and expressiveness.** Event-driven services connected to an even-based system environment express their capabilities and interests in the form of subscriptions (in the case of event consumption) and advertisements (in the case of event generation). As shown in (Eugster et al., 2002) there are conflicts and trade-offs between these subscription (and advertisement) expressiveness, event safety, and the scalability of the overall architecture. The expressiveness of subscription language and event safety increases the complexity of the filters used to route events notifications (see (Mühl et al., 2006)). In the case of semantic service descriptions this problem of complexity seems more acute. An algorithm with a reduced, or at least adaptable, complexity valid for evaluating complex filter expressions is still an open research problem.
Work Objectives

- **Autonomic self-management.** The effective configuration and management of a service bus involves a great deal of human interaction (Chappell, 2004). Semantic services architectures are no exception; we need to build architectures closer to comply with the self-management principles (Kephart and Chess, 2003) (self-configuration, self-healing, self-organization, and self-protecting).

### 3.3 Contribution to the State of the Art

This thesis aims at giving solutions to the aforementioned open research problems. Chapter 4 Firenze Framework presents proposed solution for the objective of developing knowledge level description framework; Chapter 5 Vega presents the solution for the objective of developing an event-driven service-oriented architecture for the Semantic Web and Semantic Grid. Let us summarize in this section the main contributions to the state of the art that will be presented through this dissertation.

#### 3.3.1 Knowledge level description framework

With regard to the first objective, we identify the following advances to the state of the art:

- **A Grid specific description framework.** We propose the Firenze Framework, a description framework capable of describing Grid services and Virtual Organizations. The key extension points are:
  
  - **Non-Functional Properties.** For the Firenze Framework both types of information (functional and non-functional) are first class citizens.
  
  - **Virtual Organizations description.** The Firenze Framework provides a formal and explicit description of the institution that is created by the sum of its services; the rules that govern the interaction between the entities involved; and the entities themselves (i.e. providers, consumers, and all the other roles that may coexist in a Virtual Organization).
  
  - **Complex interactions.** The Firenze Framework is able to describe complex interactions that include multiple parties, meaning more than two cooperating participants that may participate playing different roles; multiple views, meaning multilayered interaction descriptions that different levels of abstraction; and finally, complex message exchanges, meaning interactions between two entities more intricate than just a request/response pattern.
• **Grid services instances as reactive systems specification.** Considering services as reactive systems brings the possibility of describing the lifecycle, complex behaviour and state-dependent conversations of concrete Grid services instances.

  o **A knowledge level description framework.** The Firenze Framework also offers clear separation between representational level and knowledge level. In consequence, the main roles of knowledge level modelling (identified in (Van de Velde, 1993)) remain unaltered, providing the framework:

    • **Guidance in knowledge acquisition.** The use of a model in terms of knowledge is equivalent to knowledge acquisition; the model itself can be used as a frame for this knowledge acquisition process. This role can be very useful for refactoring a pre-existing Grid system to a Semantic Grid system and building a Semantic Grid system from scratch following a knowledge acquisition process (after all, most of the semantics are domain dependant and should be provided by an expert in the field).

    • **A functional specification of system.** A knowledge level model predicts behaviour. It can be naturally interpreted as a functional specification of a system's behaviour. Current Semantic Web services description frameworks are also functional specification of a system, but they are attached to a concrete description language (as we outlined in the next paragraph).

    • **High-level system design.** As the Firenze Framework though working at knowledge level is very structured, the generated instance is a high-level design of the described system. This is a better option than handling directly the representational language is more prone to inconsistencies and errors, and the quality of the design can be affected because it is constrained by the characteristics of the chosen language (the reason for that is the rupture of the separation between the design and implementation phases).

    • **Sharing and reuse.** From a single design made using the Firenze Framework we can easily obtain different representations, in different languages, expressivities, granularities, and with different reasoning mechanisms.

3.3.2 The event-driven Semantic Service-Oriented Architecture

With respect to the second main goal our work (the event-driven service-oriented architecture) the main contribution to the state of the art is Vega, an event-driven semantic service-oriented architecture that makes effective use of the proposed description framework to
Work Objectives

provide knowledge services. The main defining characteristics of the result architecture are its change of the model on interaction; and its capacity to scale. More precisely:

- **Change of model of interaction.** Vega is the first semantic service-oriented architecture that changes the model of interaction from the request/response model to an event-driven interaction model. From this perspective this architecture can be considered as complementary to the already existing semantic service architectures. It formalizes and provides algorithms for supporting the core activities in an event-driven architecture, which are event brokering, event handling and event detection.

- **A scalable architecture.** Albeit we are dealing with a knowledge-intensive environment, Vega is able to handle a huge number of event notifications. This is possible owing to:
  - **Self-configurability and self-optimization.** After the addition/removal of a new event-driven service, or the spawn of new service instances, the configuration of the whole brokering system is carried out in a consistent manner autonomously. Moreover, each event broker in the architecture is monitorized adjusting itself to prevent overloading.
  - **Peer-to-peer knowledge-based event brokering.** Vega is embodied by a decentralized peer-to-peer event brokering network that uses knowledge about the components to route event notifications; decentralizing thus the service discovery and invocation processes.
  - **Imperfect routing algorithm.** Vega notification routing is performed by a novel imperfect expertise-based routing algorithm, imperfect in the sense that weakens filters evaluation on those brokers that sustain a higher throughput of event notifications.

3.4 **WORK ASSUMPTIONS, HYPOTHESES AND RESTRICTIONS**

Before carrying out the presented work, we assumed several facts that we list below. These assumptions help to explain the design decisions that we made.

- **Assumption 1.** We describe Grid environments that are service-oriented environments. That means that external agents wishing to interact with Virtual Organization do so by invoking the operations of services that belong to such organization.

- **Assumption 2.** Scalability problems in a service-oriented environment are not related with the number of different classes of services, but with the number of instances of these services. In other words, for each type of service we should expect a large number of services instances; and the number of types of services are several orders of magnitude smaller that the number of service instances.
Work Objectives

- **Assumption 3.** The detection infrastructure addresses the detection of composite events from the flow of high-level events generated by both agents and services that play a role in the Virtual Organization. Any activity of low-level detection, or composite event detection over event streams should be virtualized either by a service or an agent, whichever option fits better.

- **Assumption 4.** We consider services provided by computer systems with a low churning rate. That means that we assume both a reliable network and that services providers are reliable (e.g. we do not expect a mobile phone as a Grid service provider).

Once the assumptions have been identified and presented, the set of hypotheses of this thesis are described. The set of hypotheses covers the main features of the Firenze Framework and Vega Architecture are:

- **Hypothesis 1.** The use of knowledge level framework for the formal description and design of service-oriented environments enhances the quality of the design, as design and implementation phases are kept separated. In concrete, we foresee that it fosters the design reusability.

- **Hypothesis 2.** Problem-Solving Methods, abstract design patterns and ontologies are valid tools for the description of semantic Web services and semantic Grid services.

- **Hypothesis 3.** Problem-Solving Methods, abstract design patterns and ontologies are valid tools for the definition of formal and explicit models for describing the different aspects of Virtual Organizations.

- **Hypothesis 4.** Services (especially Grid services) are better described as reactive system rather than transformational systems.

- **Hypothesis 5.** Complementing semantic service-oriented architectures with event-driven principles benefits the provisioning of knowledge-intensive services.

- **Hypothesis 6.** Expertise-based delivery algorithms reduce the size of knowledge associated with the routing activity. This is especially accurate in cases where there are a huge number of services instances, which is one of our assumptions.

- **Hypothesis 7.** Allowing a certain degree of relaxation of the safety delivery condition, inner event brokers might not become overwhelmed resulting in an increase of the overall system scalability.
Work Objectives

Unluckily, we needed to consider a number of limitations, limitations that determine greatly our future research objectives. These restrictions are related to the proposed architecture, since we have to delimit somehow such daunting task.

- **Restriction 1.** As we consider high-level events in the eventflow, we consider a single viewed approach to event detection. Nevertheless, we should be aware that this approach could affect the reactivity of the overall system if we use too complex eventflow patterns.

- **Restriction 2.** The topology of the event brokering system is set in advance and fixed. The system flexibility resides in the process of configuring the brokers.

- **Restriction 3.** We just take into account two of the four classic autonomic computing, namely self-configuration and self-optimization. As we have assumed a low churning rate in the peers connected to the system, self-healing and self-protection features though desirable are less important. Nevertheless, in a real open-world deployment scenario they should be taken into account (specially self-healing).
Chapter 4  Firenze Framework

In this chapter we present the Firenze Framework, a knowledge level description framework rich enough to express the semantics required for describing services (both semantic Web and semantic Grid services) and Virtual Organizations. This description framework provides ontologies and representation models to express the semantics required to improve services discovery and (semi) automatic composition; and Virtual Organizations design and handling. Contrary to other description frameworks, its descriptions are made at knowledge level, allowing the creation of high-level reusable designs. High-level designs provide a view of how a system should work; and also make explicit the top-level components that will comprise it. This produces a stronger separation between the design-implementation phases, resulting in an improvement of the design quality. Moreover, designs developed with this framework are reusable. A design artefact formalized using the framework can be easily adapted to be applicable in similar situations and straightforwardly used to compose more complex ones; and from a single design we are able to obtain different descriptions formalized in different implementation languages.

Firenze Framework is a theoretical description framework to formalize and design services and Virtual Organizations in a formal fashion at the knowledge level. Its main hypotheses are that Problem-Solving Methods, abstract design patterns and ontologies are valid tools for the creation of formal and explicit models for describing the different aspects of services and Virtual Organizations. In this chapter we describe this framework, which is composed by a stack of models that complement each other in formalizing all Virtual Organizations and services features from different perspectives. The entities that are described by these models are further classified in submodels and specifications in order to provide very structured knowledge descriptions. In order to describe such entities, each model is composed by an ontology that describes each of the entities covered by the model, and where applicable, a set of abstract design patterns.

We have identified as necessary four different layered models, namely Knowledge Representation Model, Problem-Solving Methods Model, Service Model, and the Virtual Organizations Model (we represent them in Figure 4-1). For each model we have identified the following requirements:
Firenze Framework

- **Knowledge Representation Model.** This model should provide basic building blocks for the rest of Firenze Framework models. It should provide the knowledge representation components and primitives of traditional ontology languages that could be needed in the definition of the rest of the models that compose the framework.

- **Problem-Solving Methods Model.** This model should grant the knowledge elements necessary to define Problem-Solving Methods. The provided Problem-Solving Methods characterization should be usable for describing functional features of an operation or process; and also its goals and the intentions that the invoker might have when invoking such functionality. These descriptions should be shareable and reusable; which implies that these functional descriptions should separate abstract procedural knowledge from its domain of application. Finally, despite these functional descriptions are meant to be as generic as possible, this model must also provide domain factual knowledge and a mechanism to map it to the Problem-Solving Method descriptions. It is so because the services operations are specified using these descriptions and they are domain-dependant.

- **Services Model.** The Services Model should be able to describe isolated services both statically and dynamically.
  - The static view of services includes the formal and explicit description of both the service operations non-functional and functional properties; being the later formalized by means of the Problem-Solving Method descriptions provided by the Problem-Solving Method Model.
  - The dynamic view involves the definition of long-lived conversations related with the service complex external and internal behaviours. Service descriptions should take into account state even for the external observer, as the discovery, creation and invocation of transient a Semantic Grid service’s instances depends on such information.

- **Virtual Organization Model.** The Virtual Organization Model should be complex enough for representing coalition of organizations that pool heterogeneous computational resources to achieve common objectives that a Virtual Organization represents. This model should be rich enough to describe:
  - The organizations are the entities that together provide computational resources and services. These organizations descriptions must include their descriptive information, the set of services and virtualized computing resources that each of them shares, the hierarchy of roles that the individuals of the organizations might play, its computational resources access policies, etc.
The external entities that are allowed to interact with the Virtual Organization; and its more important characteristics, roles, and privileges in the pool of shared resources.

The high-level interactions that happen in the system. The external entities, individuals that belong to the cooperating organizations and its services collaborate to achieve their goals using resources of the Virtual Organization. The definition of such choreography involves the definition of complex interactions, with multiple cooperating participants, each of which should be characterized in a formal way; and multiple views, with multiple abstraction layers from a global perspective to the local perspective; and multiple stages.

The high-level happenings of interest in the system that represent the global state of the Virtual Organization as a whole, and the needed representation mechanisms.

The domain-independent functionality spreads across the system that the Virtual Organization provides; that should be seamlessly incorporated to its belonging services.

The rationale of building the Firenze Framework as a layered model is summarized as follows:

- **Layers are built one on top of the other.** Each of the layers depicted in Figure 4-1 is built using the elements that belong to its underneath layers. In other words, the top-level components that constitute the upper layers are built using components that belong to the lower layers.
Firenze Framework

- **Layers can be selected at ease.** As we depict in the picture, the framework can be used to describe different entities: Problem-Solving Methods alone, Semantic Web services, and finally, the whole stack of models can be used to describe Virtual Organizations (and their associated Grid services). The decision of which layers to pick up is made on the basis of the artefacts that the user wishes to design. The labelled arrows that we include in Figure 4-1 represent the possible choices.

Most of the content of this chapter is devoted to describing each of the layers of the framework. As depicted in Figure 4-1, on the top of the stack we place the *Virtual Organizations Model*, model that describes the institution that is created as the sum of services provided by different organizations and the rules that govern the share of resources and the interaction between the entities involved (section 4.4 Virtual Organization Model); beneath this model we place the *Services Model*, which describes the upper-level concepts that define the features of a service (section 4.3 Services Model); the third model is the *Problem-Solving Methods Model*, which describes the elements that represent both the internal structure, the functional features, the intention of a service and the domain in which the service will be used (and, consequently, the domain of the Virtual Organization)(section 4.2 Problem-Solving Methods Model); and finally, the *Knowledge Representation Model* specifies the knowledge representation entities used as building blocks for all the aforementioned models (section 4.1 Knowledge Representation Model).

In the following sections we thoroughly illustrate the Firenze Framework models using a bottom-up approach. In order to describe each of the models we need to make references to elements of the other models underneath in the model stack. For the sake of clarity we use different namespace prefixes, namely *kr*: for the Knowledge Representation Model; *psm*: for elements that belong to the Problem-Solving Methods Model; *s*: for concepts of the Service Model; *vo*: for concepts that belong to the Virtual Organization Model.

After describing each of the models in the Firenze Framework models stack, we depict the overall description and annotation process where we think that the description framework fits and it has been applied (section 4.5 Firenze Framework Design and Annotation Process).
4.1 Knowledge Representation Model

The Knowledge Representation Model specifies the basic knowledge representation components and primitives that are used to define the rest of the models and ontologies that compose the framework. The Knowledge Representation Model is based upon the Internal Representation used in the conceptualization activity of METHONTOLOGY (Fernández-López et al., 1999)(Gómez-Pérez et al., 2003); and in the WebODE ontological engineering workbench knowledge model. The conceptualization organizes and structures the acquired knowledge using internal representations that are independent of the implementation languages and environments (Fernández-López et al., 1999). We propose to use these internal representations because as was portrayed in (Gómez-Pérez et al., 2003):

- These representations bridge the gap between the expert’s perception of the domain and its implementation in a representational language.
- METHONTOLOGY internal representations enable on the one hand conceptualizing the main knowledge representation components of the traditional ontology languages. On the other hand, as (Corcho, 2005) presents, these conceptualizations can also be effortlessly translated with a minimum knowledge lost using description logics such as OWL (OWL, 2004)) and RDF(S) (RDFS, 2004).

In the section 8.1 Appendix I: Knowledge Representation Model of the Chapter 8 Appendixes we include all the elements of the METHONTOLOGY Internal Representation with just slight modifications; along with graphical representation that we have defined to represent it.

4.2 Problem-Solving Methods Model

The Problem-Solving Methods Model defines the knowledge elements necessary to define Problem-Solving Methods. As we will describe in the following sections, Problem-Solving Methods are used by other models situated on over this model in the Firenze Framework stack to represent the internal structure, the functional features, the intention of a service, and the domain in which the services are used expressed. A Problem-Solving Method is defined as a domain-independent and knowledge level specification of a problem solving behaviour, which can be used to solve a class of problems (Motta, 1999). Configured Problem-Solving Methods become those Problem-Solving Method that have been further assembled and are related (and therefore only applicable) to a concrete domain.
The concept of Problem-Solving Method is present in different forms across several knowledge engineering frameworks such as KADS (Schreiber et al., 1993), CommonKADS (Schreiber et al., 1999), Components of Expertise (Steels, 1990), GDM (Terpstra et al., 1993), Generic Tasks (Chandrasekaran et al., 1992), etc. We propose to apply Problem-Solving Methods (Benjamins and Fensel, 1998)(Benjamins et al., 1999) for modelling services since they decouple the functional features of a service from its internal specification (Gómez-Pérez et al., 2004a). As presented in (Crubézy and Musen, 2004) unlike other types of formalisms such as rules and axioms, Problem-Solving Methods abstract and isolate procedural knowledge away from domain knowledge, making the behaviour of a system and the role of domain knowledge in that system explicit and easy to adapt. As described in (Chandrasekaran et al., 1998), in order to define a problem-solver we need two types of knowledge, namely domain factual knowledge, which is knowledge about the objective realities in the domain of interest; and problem-solving knowledge, which represents knowledge about how to achieve various goals. This knowledge can be in the form of a Problem-Solving Method, becoming thus descriptions in a domain-independent manner of how a class of goals can be accomplished.

Over the years, the knowledge-engineering community has made great use of Problem-Solving Methods, both for general purpose system description in the aforementioned knowledge engineering frameworks or for specific high-level tasks such as classification (Clancey, 1985), diagnosis (Benjamins, 1993), planning (Valente et al., 1998)(Tu et al., 1992), and design (Motta, 1999). The proposed ontology for the description of Problem Solving-Methods is based on UPML. UPML is a refinement of the CML (Conceptual Modelling Language) (Schreiber et al., 1994) of commonKADS (Schreiber et al., 1999)) and it was developed in the context of the IBROW project with the aim of enabling the (semi) automatic reuse and composition of Problem-Solving Methods distributed throughout the Web.

In the Firenze Framework we use Problem-Solving Methods for describing the functionally of services operations. To each service operation we associate a configured Problem-Solving Method, configured since the operation of a service is domain dependent. The advantages of using Problem-Solving Methods are the following:

- **Complex descriptions composeability, reusability, and shareability.** Each of the knowledge-components that compose Problem-Solving Method descriptions are created independently and in a domain-agnostic fashion (with the exception of domains, as seem

---

5 This objective is very similar to that of composing services; thus, we can consider that the IBROW project highlights the close relation between Problem-Methods and Semantic Web services, as it is clearly depicted in (Benjamins, 2003).
obvious). This approach greatly fosters composeability and reusability. As described in (Crubézy and Musen, 2004), Problem-solving Methods become ready-made software components that can be assembled with domain knowledge bases to create application systems. Moreover, as we use adapters following the approach described in (Fensel, 1997), Problem-Solving Methods, and its components, can be organized in libraries using meaningful taxonomies, improving thus their shareability and reusability.

- **Knowledge-base service discovery.** Associating services operations with Problem-Solving Methods provides an explicit and formal terminology that allows us to index and retrieve operations in a meaningful fashion. Services operations discovery becomes thus multifaceted, as they can be discovered taking into account details of the domain where the service is used, the intended functionality of the service operation, and even just considering the objectives without prior awareness of the operation existence (what is referred as capability-driven service invocation (Motta et al., 2003)).

![Figure 4-2 Problem-Solving Methods Model main components.](image)

In Figure 4-2 we represent the Problem-Solving Methods Model main components. Before we start describing thoroughly each of these components in the following sections, let us describe them in a high-level and didactic manner in order to put the reader in context. As we have illustrated, a configured Problem-Solving Method provides a knowledge level specification of a process, and it is constructed putting together several knowledge components (as represented in Figure 4-3).
These components, namely goals, tasks, methods, and domains, are created and described separately. When needed, they are assembled together (by means of bridges, see Figure 4-3) to create such formal processes descriptions. But apart from reusability in the design phase, it is important to realize that these knowledge components are also extremely useful once the Problem-Solving Method has been configured. Despite being related with each other and grounded to a specific domain, each of these components still provides a different perspective of the process being described. The task describes what the process does as a black box, relating inputs with outputs and depicting its applicability. The method describes how the functionality is achieved by the process, the reasoning steps and types of knowledge needed to perform the functionality described by the task; the goal describes why the process should be performed, specifying which will the future state of affairs be once the process has been executed; and finally, the domain that describes where the process can be applied, the application-domain of the process. Finally, adapters are the glue that brings together the knowledge components, bridging the possible existing gaps.

As we have stated, this Problem-Solving Methods Model that we propose is based on the UPML model. Its main differences:

- UPML tasks versus Firenze Framework tasks and goals. A UPML task description also specifies goals that should be achieved in order to solve a given problem. In the Firenze Framework, as we have already stated, we separate between tasks and goals. Tasks henceforth solely describe functionality, and the objectives and user-desires are all
contained in goals. As we describe in section 4.2.4 Goals, describing goals separately as first-class citizens fosters the indexation according capability, and promotes terminological and functional decoupling.

- **UPML methods versus Firenze Frameworks.** In the first place, there is a nomenclature difference. Firenze Framework methods correspond to the UPML Problem-Solving Method concept; the Firenze Framework concept of Problem-Solving Method is the construct formed when configuring a task together with a method. Regarding the nature of the descriptions contained in methods there is also a great difference. We propose a more complete and structured definition of methods. Whereas UPML does not impose (neither propose) a language for describing the complex structure of composite methods; the Firenze Framework provides a complete, yet made at knowledge level, description of the how a complex method coordinates and combines other Problem-Solving Methods to achieve its functionality.

Let us describe in detail each of the main elements of the Problem-Solving Methods Model that we represent in Figure 4-2 in the following sections.

### 4.2.1 Tasks

A *task* describes an abstract and domain independent operation to be solved; they embody the operation functional description in a completely domain independent manner. They specify both the input/output parameters of the abstract operation (in the form of input/output roles) and its competence. Thus, in accordance we divide tasks definitions in two parts:

- **Interaction definition.** It describes the input roles and output roles of the task. Role types are concepts and/or attributes of concepts that belong to the task ontologies (i.e. those ontologies related with the task with the `usesOntology`).

![Figure 4-4 Task description main concepts and relationships.](image-url)
Firenze Framework

- **Competence definition.** The competence of a task is a set of logical expressions that describe its applicability and functionality. Preconditions and postconditions constraint and make claims about the inputs and output roles of the task. **Preconditions** specify the state in which the inputs should be in order that the task could be applicable; **postconditions** specify the state of the outputs after carrying out the task, and the relationship between output roles and the input roles. The competence of a task also contains applicability conditions and effects, which are logical expressions about the real world state. The **applicability conditions** specify in a formal fashion the state of the world in which is relevant or applicable the task; and the effects specify the change in the world consequence of the application of the task.

In order to delimit and make explicit the vocabulary that is used, each task is associated with the set of ontologies used to define the task. These ontologies contain the vocabulary for specifying each of the expressions of the aforementioned models. As shown in Figure 4-4 it is represented by means of the `usesOntology` term relation. Please note again that tasks are defined in a domain-independent manner, this ontology will be mapped to a concrete domain one the Problem-Solving Method is configured. As this description is also independent from the method that solves the task, the same applies to the ontologies used to describe methods.

4.2.2 Methods

Methods detail the abstract domain independent reasoning process to perform a task. As described in (Fensel et al., 2003), they describe the reasoning steps and types of knowledge needed to perform a task. Therefore, the method embodies the operational specification of a Problem-Solving Method. We distinguish between two types of methods: **primitive methods**, those that solve directly the task in a single step; and **complex methods**, which are those that do not solve the task directly, but provide a formal description of both the decomposition of the task into sub-tasks and the coordination of those subtasks.
As portrayed in Figure 4-5, primitive methods and complex methods are described differently. The primitive method is described in an analogous manner to tasks, dividing its definition in two parts.

- **Interaction definition.** It describes the input and output roles of the primitive tasks. The role types are concepts and/or attributes of concepts restricted to those that belong to the method ontologies, as we describe later on.

- **Competence definition.** The competence of a primitive method is a set of logical expressions that describe its applicability and functionality. Preconditions and postconditions constraint and make claims about the inputs and output roles of the method. Preconditions specify the state in which the inputs should be in order that the method could be applicable; postconditions specify the state of the outputs after carrying out the method, and the relationship between output roles and the input roles. The competence of a method also contains applicability conditions and effects, which are logical expressions about the state real world. The applicability conditions specify in a formal fashion the state of the world in which is relevant or applicable the method; and the effects specify the change in the world consequence of the application of the method.

Complex methods describe how to decompose a task into several simpler subtasks in order to achieve the functionality described by the task. We divide this decomposition in two different specifications, namely the method knowledgeflow specification and the controlflow specification.

- **Knowledgeflow specification.** The knowledgeflow specification of a method defines the data interchange among the tasks in which a given composite method has been decomposed, depicting how their inputs/outputs are related and the type of the information that they share. Each knowledgeflow model is associated with a set of internal roles that store the
Firenze Framework

internal intermediate results shared among different tasks. In some sense, the knowledgeflow of a method orchestration can be seen as a semantically enriched Data Flow Diagram. The inputs and outputs roles are not much different from these internal roles; let us depict it graphically in Figure 4-6.

![Figure 4-6 The inputs and outputs of a complex method in relation with its knowledgeflow.](image)

- **Controlflow specification.** The controlflow specification describes the process carried out when performing a method. We propose a minimal set of programming primitives to describe the operational description of a composite method. In order to describe their use, we propose the use of a set of abstract design patterns. A pattern, in the software engineering field, is defined as the abstraction from a concrete form that keeps recurring in specific non-arbitrary context (Riehle and Züllighoven, 1996)(Gamma et al., 1995). They provide independence from the underneath implementation technology and independence from the essential requirements of the domain that they are attempting to address. We follow an approach similar to the work carried out by van der Aalst and colleagues (van der Aalst et al., 2003), later updated in (Russel et al., 2006), in the description of workflows using patterns. Their approach has proven that abstract design patterns offer an effective way for workflows descriptions; patterns can be used as templates to aid the designer to generate new processes descriptions. Abstract design patterns also provide a catalogue of common situations that can be used by a knowledge expert in order to improve the knowledge acquisition process. In fact, the first set of abstract design patterns that we are going to present, the controlflow patterns, are a subset of those proposed in (van der Aalst et al., 2003). We describe the set of patterns, providing for each of the patterns a description of the pattern and its context; its graphical representation, which is used to define the models; and synonyms, which point to operators and constructs of different related languages that are used in the domain where the patterns were extracted.
4.2.2.1 Controlflow Patterns

In Figure 4-7 we depict the set of controlflow abstract design patterns that we propose for defining the orchestration of complex methods, which are a subset of the ones defined in (van der Aalst et al., 2003). We have selected those that we consider the most important and relevant for the high-level description of processes and we have adapted their description to align it with the Firenze Framework vocabulary (some of the original patterns that appear in (van der Aalst et al., 2003), but specially those that appear in the extension of the original set in (Russel et al., 2006) describe too low level situations that are not appropriate for a knowledge level description framework). Finally, for each of these patterns we have proposed a graphical representation.

Figure 4-7 Controlflow patterns catalogue.

We distinguish between basic controlflow patterns, that depict situations that are elementary on any workflow system; branch and synchronization controlflow patterns, that extend some of the basic controlflow patterns making them able to run parallel branch of execution; and structural controlflow patterns, that depict how to define loops and how to define the end of the execution of a controlflow description.

---

6 We omit the synonyms of each pattern, as this information can be found in (van der Aalst et al., 2003).
4.2.2.1.1 Basic Controlflow Patterns

As depicted in (van der Aalst et al., 2003), the basic controlflow patterns are the elementary controlflow concepts that were defined by the WfMC (Workflow Management Coalition) (WFMC, 1999). As portrayed in Figure 4-7, the basic patterns are the sequence controlflow pattern, parallel split controlflow pattern, synchronization controlflow pattern, simple merge pattern and exclusive choice controlflow pattern. We shall describe each of them in the following sections.

4.2.2.1.1.1 Sequence Controlflow Pattern

Description: A task in controlflow should be carried out after the completion of another task.

Graphical representation: Figure 4-8 graphically depicts the sequence controlflow patterns. If a task_1 has to be followed by task_j, we simply relate them with an arrow.

![Figure 4-8 Sequence controlflow pattern graphical representation.](task_i → task_j)

4.2.2.1.1.2 Parallel Split Controlflow Pattern

Description: There is a point in the execution where the path of execution splits into multiple threads of control that can be executed in parallel, thus allowing different tasks to be executed simultaneously.

Graphical representation: In Figure 4-9 we depict this pattern. After the task_i is carried out, the execution flow is separated in several paths (from two to several). In the picture we represent the first step of two of these paths, task_j and task_k, which means that after them there can be any number of task or controlflow constructs.

![Figure 4-9 Parallel controlflow patterns graphical representation.](task_i → task_j → task_k)

---

7 http://www.wfmc.org/
4.2.2.1.1.3 Synchronization Controlflow Pattern

**Description:** At a given time in the controlflow, multiple parallel execution paths converge into one single thread of control. The execution is halted until all the parallel execution paths reach that point, becoming thus synchronized.

**Graphical Representation:** In Figure 4-10 we depict the synchronization controlflow pattern. In the picture we represent the set of converging control paths using their last task (e.g. \textit{task}_i, ... \textit{task}_j). Once all of these tasks have been carried out, the execution continues (we represent the first step of this flow as \textit{task}_k).

![Figure 4-10 Synchronization controlflow pattern graphical representation.](image)

4.2.2.1.1.4 Exclusive Choice Controlflow Pattern

**Description:** There is a point in the controlflow where, based on a decision expressed using a Boolean condition, one (and only one) of several branches is chosen and executed. Which path is chosen depends on \textit{condition}_j, ..., \textit{condition}_k. Note that these conditions must be mutually exclusive in order to prevent that two or more controlflow paths are triggered. The following condition must hold:

\[
\text{condition}_j \land \ldots \land \text{condition}_k \models \emptyset
\]

**Graphical representation:** Figure 4-11 depicts the pattern. After the \textit{task}_i is carried out, the execution flow is separated in several paths (from two to several). In the picture we only represent the first step of two of these paths, \textit{task}_j and \textit{task}_k, after which there can be an arbitrary number of task or controlflow constructs.

![Figure 4-11 Exclusive choice controlflow pattern graphical representation.](image)
4.2.2.1.5 Simple Merge Controlflow Pattern

**Description:** A point in the controlflow where two or more alternative branches come together without synchronization (contrary to 4.2.2.1.3 Synchronization Controlflow Pattern). Nevertheless, this pattern assumes that none of the alternative incoming controlflow paths are ever executed in parallel. So after that point it is guaranteed that there will be just one running controlflow path.

**Graphical Representation:** In Figure 4-12 we depict the simple merge controlflow pattern. In the picture we represent the set of converging control paths using their last task (e.g. \(task_i\) … \(task_j\)). Once the execution of one of the incoming paths arrives to the merging point, the execution goes on (we represent the first step of this merged controlflow \(task_k\)).

![Simple merge controlflow pattern graphical representation.](image)

4.2.2.1.2 Advanced Branching and Synchronization Controlflow Patterns

Advanced branching and synchronization patterns are those that reflect more complex situations, generally those that imply the existence of multiple instances of a process running simultaneously. We have chosen just two of the original patterns depicted in (van der Aalst et al., 2003), since the rest of them were too specific for the high-level description that we desire for the description of methods. We have only chosen two that complement those that we depicted as basics, the multi-choice controlflow pattern and the multi-merge controlflow pattern.

4.2.2.1.2.1 Multi-choice Controlflow Pattern

**Description:** There is a point in the controlflow where, based on a decision expressed using a Boolean condition, several branches are selected to be executed. Which path is chosen depends on \(condition_j\)…\(condition_k\). Contrary to the case of the pattern 4.2.2.1.2.1 Multi-choice Controlflow Pattern these conditions may overlap, which results in several paths of execution running in parallel.

**Graphical Representation:** Figure 4-13 depicts the multi-choice controlflow pattern. After the \(task_i\) is carried out, the execution flow is separated in several paths (from two to several). In the
picture we represent the just first step of two of these paths, \( task_j \) and \( task_k \), after which there can be an arbitrary number of task or controlflow constructs.

![Diagram of controlflow pattern](image)

Figure 4-13 Multi-choice controlflow pattern graphical representation.

4.2.2.1.2.2 Multi-merge Controlflow Pattern

**Description:** There is a point in the controlflow where two or more branches converge without synchronization. For every activation of every incoming branch, one outgoing branch gets activated.

**Graphical Representation:** In Figure 4-14 we depict the synchronization controlflow pattern. In the picture we represent the set of converging control paths using their last task (e.g. \( task_i \)) \( \ldots \) \( task_k \). Once the execution of one of the incoming paths arrives to the merging point, the execution goes on (we represent the first step of this flow \( task_k \)).

![Diagram of multi-merge controlflow pattern](image)

Figure 4-14 Multi-merge controlflow pattern graphical representation.

4.2.2.1.3 Structural Controlflow Patterns

*Structural controlflow partners* describe loops and define how the termination of a process is indicated. We will use again patterns defined by van der Aalst and colleagues. We propose to use the arbitrary cycles controlflow pattern (similar to the one defined in (van der Aalst et al., 2003) but we force the definition of a Boolean condition) and the implicit termination controlflow pattern, which simply states that there is no explicit symbol to define the end of a process.
4.2.2.1.3.1 Arbitrary Cycles Controlflow Pattern

**Description:** This pattern represents the situation where part of the orchestration keeps on repeating if or while a condition holds.

**Graphical representation:** In Figure 4-15 we depict two different possibilities for this pattern, depending on whether the condition is checked after/or before the first iteration. The dotted box represents the block to be repeated, we have included some tasks (represented as $task_0^{...}task_N$), but this block can include any controlflow description.

![Figure 4-15 Arbitrary cycle controlflow pattern graphical representation.](image)

4.2.2.1.3.2 Implicit Termination Controlflow Pattern

**Description:** We do not define any special controlflow construct to define the end of a process. A controlflow definition is considered finished when simply there is nothing else to be done.

**Graphical representation:** This pattern asserts that there will not be an specific symbol for representing the end of a path of execution. Therefore, as seems pretty obvious, we do not propose any graphical representation for it.

4.2.3 Domains

The knowledge components that we have portrayed in the Problem-Solving Methods Model describe domain-independent entities. The domain to the contrary introduces domain-specific knowledge in the equation, containing the information objects of a concrete application field. The domain model is related to tasks and methods by means of adapters (task-domain bridges and method-domain bridges respectively) in order to represent a concrete description of an operation in a concrete domain. The elements that define a domain reflect those defined for a domain in UPML (Fensel et al., 2003), consisting of a set of assumptions, domain ontologies,
knowledge base, properties, and as the rest of Firenze Framework referenceable components, is also associated with a meta-knowledge term. More precisely:

- **Properties.** Properties are logical statements that categorize the domain knowledge. They directly derive from the domain that they describe; they are introduced with the purpose of a better formalization of the domain.

- **Assumptions.** Assumptions are logical statements that categorize the domain knowledge, but contrary to properties, they are introduced whilst building the domain model from the real world. They do not stem logically from the domain; instead they are introduced by the knowledge modeller, and as such should be taken for granted.

- **Domain ontologies.** The domain, as all the knowledge components of a Problem-Solving Method, is formalized using a set of ontologies.

- **Knowledge base.** The knowledge base contains the instances of the concepts of the domain ontologies that describe the individual of the domain.

![Figure 4-16 Domain description main concepts and relationships.](image)

Figure 4-16 shows the elements that represent a domain. The Domain concept (described by a Meta-Knowledge Term) is related using different term relations (hasAssumptions, hasProperties, usesOntology and hasKnowledgeBase) to represent its assumptions, properties, ontologies and knowledge bases respectively.

### 4.2.4 Goals

A goal describes pursued states; which as described in (Weiss, 2000) are formally described logically consistent and achievable desires. We adopt an explicit goal-driven approach in the sense that, unlike the initial definition of Knowledge Level, goals are separated from the body of knowledge that represent the state of affairs in the environment. We adopt a goal-driven approach because as depicted in (Russel and Norving, 2009) such approach is well suited in knowledge intensive situations. This model of behaviour implies the formal and explicit
Firenze Framework

description of the information that supports agent decisions, as it is of great importance that an agent is aware of the desirable future states apart from having knowledge about the current state of affairs (which is the agent body of knowledge).

- **Indexation according to its capability.** Problem-solving knowledge may be indexed by the goal it serves; it facilitates the search of Problem-Solving Methods in terms of external agents needs.

- **Decoupling.** Apart from the ability of formalizing intention, from a more pragmatic view, the separation between goals and the actual functional description of the underlying system capability (embodied as configured Problem-Solving Methods) permits two forms of decoupling that we believe of great importance, specially when describing knowledge-intensive distributed systems. They are:

  - **Terminological decoupling.** External agents (e.g. client side users, software agents, etc…) exist in specific terminological context that may not be the same as the context where the system description is made. An external agent set of goals is likely to be expressed in its own context vocabulary using different terms or ontologies from those used to describe the functionality of the system.

  - **Functional decoupling.** An external agent described in terms of its intention is completely decoupled from the description of how a concrete system facilitates the transition to the desired state of affairs. On the one hand the external agent is not aware of the whole system functional capability. On the other hand, changes made on the system are not propagated to the agent description. Agents are likely to interact with more than one system, and systems are likely to evolve; nevertheless their description should be invariant since conceptually the agent and its intentions remain the same.

![Figure 4-17 Goals and tasks relationship.](image)
Goals are related with configured Problem-Solving Methods by means of adapters that define assumptions (teleological assumptions according to the taxonomy based on the specialization organization of assumptions described by (Benjamins and Pierret-Golbreich, 1996)). We will describe this topic in detail when we describe adapters in the next section, 4.2.5 Adapters.

Figure 4-18 contains the main elements that we consider when describing a goal. A goal is defined as the disjunction of a set of logic expressions (which are embodied as a group of Expression concepts)

4.2.5 Adapters

Reusable knowledge components (i.e. tasks, methods, domains, and goals) are specified independently from each other, and in consequence there is a need to introduce components to glue and bring together the knowledge components that compose a configured Problem-Solving Method. We refer to such components as adapters, and also are used to bridge the possible gaps between the configured Problem-Solving Methods and the domain and the goals of the external agent.

Adapters (Fensel and Groenboom, 1997) enable the independent specification of problem definitions. This definition, somehow too general, is further specified in (Fensel, 1997) where adapters gained their status as first class knowledge components. Fensel justifies the importance of adapters in the construction of knowledge-based systems from reusable elements requires, on the fact that adapters adjust these reusable elements to the application-specific circumstances. In the Firenze Framework the main uses of adapters are to foster reusability and to introduce assumptions. More precisely:

- **Ontology-based reuse of knowledge components.** This use is basically the one shown in the introduction to this section. Adapters contain mappings relations that bridge the conceptual and syntactic gaps between the ontologies used to define each of the main Problem-Solving
Method knowledge components. Therefore they make explicit relationships between the ontologies of task, method and domain; relationships which are assembled for a concrete and specific knowledge application (as depicted in (Crubézy and Musen, 2004)).

- **Assumptions definition.** The definition of assumptions in the Firenze Framework is contained as part of the definition of adapters. We consider two types of assumptions:
  
  - **Ontological assumptions.** These assumptions pose requirements on the domain where the Problem-Solving Method is to be applied. These assumptions, as we shall see, are contained in task-domain bridges and method-domain bridges.
  
  - **Teleological assumptions.** These assumptions impose functional restrictions. As described in (Benjamins and Pierret-Golbreich, 1996) these assumptions can be used to reduce the complexity of solving a given goal by specifying and restricting the situations where the Problem-Solving Method can effectively satisfy a given goal. More generally, those assumptions restrict the circumstances under which a given configured Problem-Solving Method can be used to achieve certain goal. These assumptions, as we shall see, are contained in goal-PSM bridges.

Figure 4-19 depicts graphically the kind of assumptions which are contained in each of the adapters of our framework in accordance to the afore shown assumptions diagram.

As shown in Figure 4-20 in our framework we contemplate two types of adapters, namely refiners and bridges (we use the same terminology defined in UPML (Fensel et al., 2003)). Let us describe this topic more thoroughly.
Reiners. Refiners link knowledge components of the same nature, specializing knowledge components into more concrete ones. As described in (Fensel, 1997) they can be used for stepwise refine generic Problem-Solving Methods into more specific ones, which is very useful for navigating Problem-Solving Methods libraries. In the Firenze Framework we contemplate the definition of refiners for each of the main knowledge components (i.e. tasks, methods, domains, and goals).

Bridges. Bridges link knowledge components of different nature. Depending on the components they bridge, in this thesis we distinguish the following subtypes of bridges:

- **Goal-PSM bridges.** As presented in the section devoted to goals, they represent intentions and future states of affairs. On the one hand, Goal-PSM bridges solve in an explicit manner the possible terminological discrepancies between the vocabularies used to express agent intentions and the Problem-Solving Method domain terminology. On the other hand, Goal-PSM Bridges pose additional restrictions over those goals, so that we can assure that concrete Problem-Solving Method might be able to satisfy it (i.e. the teleological assumptions that we have already discussed and represented graphically in Figure 4-19).

- **Task-method bridges.** As we described tasks and methods are defined in an independent manner, in the shake of reusability. When configuring Problem-Solving Methods tasks and methods should be related, which is exactly the role of Task-Method bridges. It is important to realize that the functionality depicted by a task may be achieved applying different methods; and vice versa, the operational specification that a method represents might be associated with several tasks (the latter case seems less intuitive but might be possible).

- **Task/Method-domain bridges.** Both the functional specifications embodied as tasks, and the operational description contained in methods definitions were made in a domain independent manner. Task/Method-domain bridges map the terminologies of tasks and methods into terms of the domain ontologies. Any restriction on the domain when
defining these bridges forms part of the ontological assumptions, as we have already defined and represented graphically in Figure 4-19.

![Diagram](image)

Figure 4-21 Adapter description main concepts and relationships.

Figure 4-21 shows the main knowledge components that we propose for describing adapters. Note that refiners and the different types of adapted are described identically, the difference relies in the nature of the elements that they relate. Adapters are formed by a set of mappings (instances of the Mapping concept related by the contains relationship). Those mappings are defined among conceptualization elements that belong to the description of the related knowledge components (both instances of the general Knowledge Component, and related by means of the hasOrigin and hasDestination term relations). In cases where we are describing a bridge, the types of the related knowledge components would be different, whereas in the case of refiners, the related knowledge components are of the same type.

### 4.3 Services Model

The Service Model describes the upper-level concepts that define the features of a service. We consider services in a broad sense, a collection of functional capabilities, which are offered remotely by a set of network accessible operations. From this rather generic definition we can derive many more specific definitions of services taking into account their characteristics, intended functionality and supporting technology. We just limit the technological perspective, as we have already stated several times, we address two types of services, Web services and Grid services.

State of the art description approaches of services, such as WSDL, WSMO, OWL-S, etc. depict services as atomic and transformational systems. A transformational system exists solely for transforming an input into an output. After that, the system ends either returning the result or notifying an exceptional situation that impeded its normal operation. This functional approach
makes sense exclusively in the case of Web services, since they were considered stateless interfaces of a set of internet-accessible operations.

A reactive system (according to (Wieringa, 2002)) is a system that, when switched on, is able to create desired effects in its environments by enabling, enforcing, or preventing events in the environment. They are described in terms of their ongoing behaviour; they do not terminate. The rationale and implications of considering services as reactive systems, according to the reactive system characteristics described among others in (Wieringa, 2002)(Aceto and Larsen, 2005), are the following:

- **Changes in service duration and nature.** Services like Grid services or any other that implies a complex process enactment, may take a long time to terminate. They are far from being atomic taking into account that from the point of view of an external observer it takes a long time from its beginning until its end. Furthermore, different stage points of interest might be defined across the execution of the service.

- **Explicit external state.** The interaction with Grid services, contrary to Web services, is stateful, and it depends on the observable state of the service. Thus the current external state of a service is stateful and as such should be represented and accessible to external entities.

- **The service continuously interacts with its environment.** Services keep on receiving inputs and sending the corresponding outputs in a loop. Thus, unlike transformational systems they do not usually terminate when they output the results of its computations.

- **Complex and concurrent interfaces.** A service may usually involve complex processes that may interact and operate in parallel; they are not simple sequential function calls executed in a chronological order. This is especially important in the Firenze Framework, since the inclusion of the aspects and facets (see section 4.4.5 Themes Submodel) relies partly on the use of concurrent service interfaces.

- **Service responses are not direct function of the requests.** In a transformational system outputs can be expressed as a function over the inputs values. The outputs of a reactive system are based both in the inputs that they receive and in the current and past internal state of the service (i.e. the state of the resources that it may handle and the state of its interactions). Moreover, they have also the ability to change this state, therefore affecting their associated resources and the environment they are in. This property is important for describing Grid services since they virtualize computational resources that generally are stateful.
In the following sections we describe in detail how we will describe services in the Firenze Framework. But, before we get into details, we present a simplified example of a service description, which will help us to contextualize each of the representation elements. In Figure 4-22 we depict an example of a service. In this picture we portrait Service$_A$ that exposes three different operations, namely, Op$_A$, Op$_B$, and Op$_C$. As you can see, the service description can be divided in three main areas, namely service behaviour descriptions, service profile description, and service internal description.

Figure 4-22 Service as a reactive system in the Firenze Framework.

- **Service behaviour description.** Service behaviour formalizes how to interact with Service$_A$. It combines the description of its external state; and the process that defines the interaction with the service. This interaction process combines the different interactions (i.e. int$_A$, int$_B$, and int$_C$) that have to be carried out to invoke each of Service$_A$ operations (Op$_A$, Op$_B$, and Op$_C$ respectively) with a set of primitives (the * and + operators which specific meaning will be described later on) that specify the order and conditions that apply to these interactions. For example, in the case of Service$_A$, it keeps interacting while cond$_i$ carrying int$_A$; and depending if either cond$_i$ or cond$_j$ hold, the service is able to perform either int$_B$ or int$_C$ respectively. These topics will be covered in section 4.3.2.2 Interaction Process.
Service profile description. The profile description of the Service\textsubscript{A} is the static semantically enriched information that is used by external agents for discovering it. The profile description of Service\textsubscript{A} is divided in its functional and non-functional profiles. For the functional description of Service\textsubscript{A}, each of its operations are related to a configured Problem-Solving Method composed by a task, method and a domain (plus the set of adapters to configure all these knowledge elements). The functional profile is solely the task attached to the domain, since they represent a formal description of what the operation does in the service domain. The non-functional profile identifies the service and provides different types of information about the service (QoS, authoring, category, availability, etc.). In the picture they are represented as a set of tuples (\textit{prop, val}). We will provide all the details about the service profile in the section 4.3.1 Profile.

Service internal description. The service internal description formalizes the orchestration of steps that an operation of Service\textsubscript{A} carries out to provide the functionality defined in Service\textsubscript{A} profile. As we shall see in forthcoming sections (and as portrayed in Figure 4-22) the orchestration process of an operation is formalized using the method of the configured Problem-Solving Method. The method attached to the service domain represents in a formal fashion the internal process carried out by the service operation in the service domain. How we relate the service with that method is described in 4.3.2.1 Orchestration Process.

Once we have informally shown how we will depict services, let us provide a more formal description. Figure 4-23 contains the top-level concepts of the service model, and how they related each other.
concept using the hasInput or hasOutput term relation). The service is externally described associating a Profile concept instance (using the presents term relation), which as we shall see in the next section, represent the functional (see 4.3.1.1 Functional Profile) and non-functional (see 4.3.1.2 Non-Functional Profile) properties of the service. We also associate the service with two different types of processes, namely the Interaction Process and Orchestration Process (using the externallyDescribedBy and internallyDescribedBy term relations). As we have already presented in the example, the Interaction Process concept represents the observable behaviour of a service (see 4.3.2.2 Interaction Process) whereas the Orchestration Process represents the internal sequence of tasks invocations that must be performed to achieve each operation functionality (see 4.3.2.1 Orchestration Process). We describe each of these topics thoroughly in the following sections.

4.3.1 Profile

The profile of a service expresses in a formal and explicit manner what the service does from two orthogonal perspectives containing both functional and non-functional properties of the service. The main purpose of a service profile, as described for other services description frameworks for example in (Klusch et al., 2005)(Keller et al., 2004), is the automatic discovery of a service. This automatic location of services involves the analysis of whether a service provides a particular functionality and that fulfil certain non-functional constraints using the information contained in the profile. Note that profiles are static, in the sense they describe what the service does and not how to interact with it.

![Figure 4-24 Service profiles.](image)

As we depict in Figure 4-24, Firenze Framework represents the two orthogonal perspectives of the profile explicitly. Therefore the top-level knowledge-component Profile is associated with Functional Profile by means of the hasFunctionalProfile term relation; and a Non-Functional Profile, by means of the hasNonFunctionalProfile term relation.
4.3.1.1 Functional Profile

The functional profile of a service specifies in a formal and explicit manner the expected results when each of the service operations is invoked. As we consider that the functionality of a service is provided by a set of network accessible operations, describing service functionality implies the description of each of the operation that the service exposes.

In order to describe each operation, the functional profile relates service operations with tasks of a configured Problem-Solving Method, since they must be grounded to a concrete domain since operation are domain dependent. The relationship is created by means of adapters of the Problem-Solving Method model (see 4.2.5 Adapters) since they can be used to group together mappings that relate each of the parameters of the operation with the roles of the task that describe them semantically. Note that in this case, adapters belong to the definition of the functional profile of the service; we reuse them for convenience and they are not part of the definition of the Problem-Solving Method.

We represent in Figure 4-25 how we formalize the functional profile of a service. Each Functional Profile concept instance is associated with a set of Adapter instances (by means of the composedOf term relation). These adapters relate Operation instances with its describing Task instances; and each of the Parameter instances associated with the operation is related with its correspondent task Role instance, using the adapter associated Mapping instances.

4.3.1.2 Non-Functional Profile

The non-functional profile of a service contains a set of attributes and parameters that measure, under different criteria, the manner in which the service implements and delivers its functionality. The non-functional profile of a service is composed by the set of non-functional...
properties and a set of concepts and relationships. The core elements of this profile are based on those presented in (O'Sullivan et al., 2005)(OWL-S, 2004). We propose the following non-functional properties, concepts and relationships:

![Diagram of non-functional profile main elements.](image)

**Identification and Description Properties.** These non-functional properties purpose is the identification and the description of a service solely in human-readable format. Thus we propose the `name`, the human-oriented name of the service; `UID`, the identifier that is unique over time with respect the host it is generated; `URI`, its unique global identifier; and `description`, a textual description of the service.

- **Service provider.** We consider the service provider as the organization that provisions the service. As shown in Figure 4-26 we represent this relationship using the `provides` term relation that links Organization concept instances (we describe them in section 4.4.6 Organizations Submodel) with instances of the Non-Functional Profile concept.

- **Service location.** Though the provider of a service has its own location, we should also define the location of the service. We consider that an organization might have several dispersed physical location. Thus we define the `locatedIn` term relation between the service non-functional profile represented in the Non-Functional Profile concept and the Locative Entity concept.
Service availability. The availability of a service can be viewed as the degree to which the service is operable and in a committable state at the start of a mission. As described in (O'Sullivan et al., 2005) availability combines temporal and locative aspects. Thus we differentiate between:

- **Temporal availability.** The temporal availability specifies when the service accepts request from requesting agents. As exposed in (O'Sullivan et al., 2005) we distinguish three types of availability, they are nominated availability, which means that service requests are accepted at/during a specific temporal pattern; negotiable availability, which means that the service requests can be configured by the requestor in conjunction by discussing temporal availability with the provider; and continuous availability, which means that the service is continuously accepting requests.

- **Location availability.** This non-functional property describes formally the places from where agents that may interact and make request to the service.

Both temporal and location availabilities can be public (service requestors are aware of these availability restrictions) or private (only the provider knows about it existence). Finally, in order to describe both availabilities we propose the use of the ontologies for availability presented in (WSMO D28, 2006) (which is itself an extension, and more importantly, a formalization of those outlined in (O'Sullivan et al., 2005)).

Service category. Services may also be organized according to any other external classifications. This is especially useful for external agents that may be aware of standard classifications. In order to allow these annotations, we foresee that a particular service Non-Functional Profile instance might be related to several Service Category instances. The Service Category concept, adapted from the one defined in (OWL-S, 2004), has the following attributes: name, the name of the actual category; taxonomy, a reference to the taxonomy scheme; and value, the assigned category in the aforementioned taxonomy.

Service properties. Apart from all the core of properties that we have defined, there are many other properties that might be of interest such as such as performance, network-related QoS, performance, reliability, robustness, scalability, transaction related properties, trust, etc. There are also some properties that may be dependent of the service domain properties. To allow the definition of this kind of service properties we will relate a service profile instance with several instances of the Service Property (based in the OWL-S Service

---

8 As we shall see, internally in the Virtual Organization we organize services in an abstract fashion classifying them in roles taxonomies.
Parameter concept (OWL-S, 2004) concept). The attributes of the Service Property are the name, the name of the property; and type, which is a reference to the type of the property (i.e. a knowledge element that belongs to an ontology).

- **Service price.** We also consider the possibility that the use of the service might have associated some cost, either monetary or a more abstract one (such as virtual currencies). We relate the profile concept Non-Functional Profile with the concept Price, by means of the hasPrize and hasCost term relations. The concept Price is left undefined, it is meant to be reified by domain specific pricing/cost concepts.

4.3.2 Service Processes

Hitherto we have just described in detail the static part of a service contract, the profile. In this section we explore the dynamic part of a service, the one that describes the external behaviour of the service and how internally the service coordinates the invocation of other services in order to achieve its functionality. We encapsulate this information in two complementary processes, namely the interaction and orchestration processes. The interaction process depicts the external behaviour of the service; other services and agents that wishing to interact with the service should be able to understand and adhere to this process. The orchestration process reflects the inner process that each of the operations of the service carry out to provide the functionality described in the profile. Let us describe both processes in a precise manner in the following sections.

4.3.2.1 Orchestration Process

By analogy with the definition of Web service orchestration presented in (WS-Glossary, 2004), the Firenze Framework considers that the Orchestration Process of a service defines the sequence and conditions in which one service invokes other services in order to realize some useful function. That is, an orchestration is the pattern of interactions that a service agent must follow in order to achieve its goal. In order to formalize this interaction we will associate to each operation with a method that solves or decomposes the task associated with the profile of the service operation.
Figure 4-27 Orchestration Process main representation elements.

Figure 4-27 depicts how the Orchestration Process relates each of the service operations with a formal description of the internal process carried out to achieve the functionality of an operation. We follow an approach for describing the orchestration process similar to the one that we proposed for the description of the functional profile of the service. An Orchestration Process concept instance can be considered as an aggregation of Adapter instances (by means of the composedOf relationship). These adapters relate Operation instances with its describing Method instance; and each of the Parameter associated with the operation instance is related with its correspondent method Role instance, using the adapter associated Mapping instances. As we did with the description of the functional profile we have to make two considerations. Firstly, the method must belong to a configured Problem-Solving Method, since service operations are domain-dependent entities; and secondly, once again we stress that this adapter is not part of the description of any configured Problem-Solving Method, but part of the description of the orchestration of a service operation.

### 4.3.2.2 Interaction Process

The Interaction Process formalizes the observable behaviour of a service. That includes the interactions it engages to achieve a goal, and the dependencies between those interactions. These dependencies may be related for example to the order of these interactions, or with the possibility of having several paths of execution. As we have already stated a service Interaction Process is created from the point of view of an external observer, and as a behavioural interface of the service, it does not describe any internal action that the service may carry out. The Interaction Process of a service provides thus a high-level description of:

- **The flow of control.** The flow of control represents the order in which the individual statements or actions of the service external interface are performed. We refer to these statements or actions as activities.
Firenze Framework

- **The execution context.** We have already stated that we describe services as reactive systems. One of the characteristics of such systems is that the Interaction Process is stateful. The different interactions and assignments of the ongoing process modify this context, and the future path of activities of this process may depend on that context.

- **The way of handling exceptional situations.** It defines how exceptional or unusual conditions that occur whilst the choreography is performed are handled. As a special type of exceptional situation we should take into account the service finalization strategies. As we have already stated, we consider service as reactive systems. As such, if they terminate after carrying out an operation, probably something went wrong.

Moreover, the description of the service interaction process should be reusable and composeable. The same interaction definition should be usable by different instances of participants that may operate in different domains. Moreover, each service process of interaction should be combinable with others; they must be reusable in different contexts.

Let us now describe how the Firenze Framework models these ideas. To begin with, we define a taxonomy of processes, being the root the *Observable Process* concept. An *Observable Process* is related with an *Activity Expression* that specifies it (by means of the `definedBy` term relation). This expression specifies the set of activities (and in which order) that the observable process performs (we define the different activities that we distinguish later on). We differentiate between three types of observable processes, namely *Interaction Process*, *Finalizer Process*, and *Exception Process*. The *Interaction Process* represents the core of the description that formalizes the observable behaviour of a service. It is related with:

- **One finalizer process.** In the model this *Finalizer Process* is related with the `hasFinalizer` term relation as depicted in Figure 4-28.

- **Zero to several exception processes.** Each of these processes handles a specific exceptional situation. As shown in Figure 4-28 these processes are triggered when certain event is received or an abnormal situation occurs. This event is formalized using an instance of *Event Expression* that is related with the `onHappening` relation with the instance of the Exception Process (event expressions are defined in the Event Submodel (see section Error! Reference source not found. Error! Reference source not found.). We also conceive the definition of an unusual condition that holds in the execution context of the process that might lead to an exception process (formalized with a *Expression* related with the process using the `onCondition` relation).
As presented in Figure 4-28 each process is also associated with its execution context, which is a set of variables and individuals that can be dynamically created, updated and queried whilst the service interacts with other services and agents. We represent it as an instance of the Knowledge Base concept (which is part of the Agent Submodel that we present in the section 4.4.1 Agents Submodel); and the hasContext term relation.

The basic building blocks of observable processes are activities (we represent their taxonomy in Figure 4-29). Activities may be simple or complex. Simple activities represent the basic unit of behaviour of a service, and they can be:

- **Silent Actions.** They describe a point of the process where the service carries out a non-observable action. That means that the service is carrying a computational activity with no visible effects to external entities to the service.

- **Assignations.** These actions represent the update of the value of one of the variables that belong to the process execution context.
Interactions. An interaction corresponds either to an exchange of information with other entity (agent or service) in order to perform a task; or to the reception/production of an event notification. In the former case, the message exchange that is produced must be specified. As depicted in Figure 4-29, we associate the Interaction concept with the ExchangeSpecification concept, which possible instances are a subset of the message exchange patterns that WSDL 2.0 (WSDL 2.0, 2007) that we consider enough for our high-level description of services. More concretely we distinguish between in-bounds message exchange patterns, which are in-only, in-out, and in-optional-out; and the out-bounds message patterns, namely out-only, out-in, and out-optional-in. We depict their meanings graphically in the picture Figure 4-30.

![Diagram of message exchange patterns](image)

Figure 4-30 Graphical representation of the message exchange patterns.

Complex Activities are defined as a recursive composition of other activities (both simple and complex). We specify them by means of activity expressions as we depict in Figure 4-29, where we define the term relation definedBy between the concepts Complex Activity and Activity Expression. As we did in the case of the description of methods controlflow (4.2.2.1 Controlflow Patterns) we propose a set of patterns that can be used as templates to aid the designer to generate complex activities descriptions.
4.3.2.3 Interactionflow Patterns

In this section we attempt to illustrate the catalogue of patterns that we have defined for describing complex interaction activities. For creating them we have followed a process similar to the one proposed by van der Aalst and colleagues (van der Aalst et al., 2003). Firstly, we have selected the main choreography description languages, WSDCL (WSCDL, 2005) and WSCI (WSCI, 2002), plus a higher level approach for describing collaboration such as the commonKADS Communication Plan Model (Schreiber et al., 1999)). We have analyzed them and taken into account the common depicted situations for creating a set of common patterns. For each of pattern we provide a description of the pattern and its context; its graphical representation; and synonyms, which point to operators and constructs of the different related languages that we have considered on its creation.

We distinguish between two main sets of interactionflow patterns, we consider the main set of them as basic interactionflow patterns, which are those that can be used to describe the core of the description of the Interaction Process; and also the branching and synchronization interactionflow patterns, which cover more complex situations with different but related branches of execution involved in the interaction process.

**4.3.2.3.1 Basic Interactionflow Pattern**

*Basic interactionflow patterns* are the core patterns for the describing interaction processes, which include situations like sequences of activities, guarded transitions, activities that run in parallel but without any interactions among them, and iterations structures. Let us describe all of them in the following sections.
4.3.2.3.1.1 Sequence Interactionflow Pattern

**Description:** An order in a pair of activities can be observed in the external behaviour of a service. Thus, an activity (previous activity in Figure 4-32) is always followed by the execution of another activity (depicted as later activity in Figure 4-32) after a given time, or immediately if the amount of time is left unspecified. Both previous and later activities can be simple or complex.

**Graphical representation:** We represent this pattern in Figure 4-32. Both activities are related with an arrow that determines the order of the observed activities. The arrow can be optionally labelled with the amount of time that exists between the end of the previous activity and the beginning of the later activity.

![Figure 4-32 Sequence interactionflow pattern graphical representation.](image)

**Synonyms or related approaches:** The situations portrayed by this pattern without the optional time parameter can be formalized by the sequence operator (both in WSCDL and WSCI) and the commonKADS SEQUENCE operator (represented with the ; symbol) has also the same meaning. The time condition can be obtained by combining other language constructs (e.g. WSCI delay).

4.3.2.3.1.2 Guarded Transition Interactionflow Pattern

**Description:** There is a point in the interaction process where before an activity (simple or complex) can be carried out certain condition should hold or some happening should be registered (what we refer as the guard). Once the condition of the guard holds, the activity can be carried out instantaneously. Depending on the nature of the guard, we distinguish between:

- **Data-driven condition.** The condition is an expression that may range over the variables of the interaction process context.

- **Event-driven condition.** The progress on the interaction process depends on the arrival on certain type of event. We represent this event as an arbitrary event expression.

**Graphical representation:** The graphical representation of this pattern is shown in the Figure 4-33.a represents the data-driven condition case and Figure 4-33.b represents the event-driven condition case.
Synonyms or related approaches: WSCDL posses a special type of activity named WorkUnit. A WorkUnit activity prescribes the constraints that have to be fulfilled for making progress and thus performing work within a choreography (WSCDL, 2005).

4.3.2.3.1.3 Parallel Interactionflow Pattern

Description: Two or more observable activities in the service external behaviour are carried out simultaneously. Those activities may be either simple or complex.

Graphical representation: Figure 4-34 shows the graphical representation of the parallel interactionflow pattern. All the activities are linked to a node that represents the pattern (which is the same as the one used for the parallel eventflow pattern as we shall see).

Synonyms or related approaches: This pattern operational semantics are identical to those of the sequence operator present in WSCI and WSCDL.

4.3.2.3.1.4 Iteration Interactionflow Pattern

Description: An activity is carried out in the interaction process several times, till a condition is fulfilled.
Graphical representation: The graphical representation of this pattern is shown in Figure 4-35, and we represent in a different fashion whether the first time that the activity is carried out only after the condition is evaluated (Figure 4-35.a) or not (Figure 4-35.b).

![Graphical representation of Iteration interactionflow pattern](image)

Synonyms or related approaches: Operators to define loops in the interaction process are present in WSCI, using the until and while choreography elements.

4.3.2.3.2 Branching and Synchronization Interactionflow Patterns

Branching and synchronization interactionflow patterns, cover more complex situations with different and relatively complex activities that may be carried out in parallel, but in some point of the interaction process are synchronized and their results somehow combined.

4.3.2.3.2.1 Switch Interactionflow Pattern

Description: There is a point in the sequence of observable activities of the service external behaviour in which one of a collection of execution branches must be chosen. This collection is formed by two or more mutually exclusive activities (simple or complex).

Graphical representation: Figure 4-36 shows the graphical representation of the switch interactionflow pattern. The collection of execution branches is represented as an unordered list of activities linked to a node labelled with a point symbol. If the decision is taken on the base of a logical expression about the execution context, we represent the pattern as shown in Figure 4-36.a; if the decision depends on the reception of an event with certain characteristics (represented as event expression, ... event expression) we represent the pattern as in Figure 4-36.b.
Synonyms or related approaches: A similar situation can be expressed in WSCDL combining the CHOICE operator and the WorkUnit mechanisms. The event-based choice (portrayed in Figure 4-36.b) is similar to the WSCI choice operator.

4.3.2.3.2.2 Choice Interactionflow Pattern

Description: There is a point in the sequence of observable activities of the service behaviour in which one of a collection of execution branches must be chosen. Unlike the afore-described switch interactionflow pattern, the condition or reasoning process that leads to election of one path over other remain unknown to the observer (i.e. it obeys to an internal evaluation process).

Graphical representation: Figure 4-37 shows the graphical representation of the choice interactionflow pattern. The collection of execution branches is represented as an unordered list of activities linked to a node labelled with the plus symbol.
**Firenze Framework**

**Synonyms or related approaches:** The situations that this pattern portrays are the same that can be formalized by means of the `CHOICE` operator (also represented with the `+` symbol) that appears in the commonKADS Communication Plan Model.

### 4.3.2.3.2.3 Synchronization Interactionflow Pattern

**Description:** If there are several paths in the interaction process going on in a parallel fashion, there might be a point where those branches need to be synchronized. The interaction is halted, till all the active incoming activities are carried out, and then the interaction keeps on normally.

**Graphical representation:** Figure 4-38 shows the graphical representation of the synchronization interactionflow pattern. There is a set of activities `activity_1...activity_j`, a subset of which may be active (i.e. activity that is being carried out at that moment). The interaction is halted in the point depicted with a circle until all these active activities finish, and then the interaction carries on, executing the next activity (i.e. `activity_k` in this case).
4.4 **Virtual Organization Model**

*Virtual Organizations* were initially considered a coalition of disperse organizations and/or individuals that share resources in a controlled fashion (Foster et al., 2001). Nevertheless, other characterizations of Virtual Organization have been proposed. Dimitrakos and colleagues (Dimitrakos et al., 2004) defined a Virtual Organization as a temporary or permanent coalition of geographically dispersed individuals, groups, organizational units or entire organizations that pool resources, capabilities and information to achieve common objectives. From an agent-oriented perspective, they are a form of dynamic service composition where a number of initially distinct entities come together, under a set of operating conditions, to form a new entity that offers a new service (Foster et al., 2004).

![Figure 4-39 Firenze Framework Virtual Organization Model.](image)

Since the emergence of OGSA (*Open Grid Service Architecture*) (Foster et al., 2002), Grid environments have been characterized as service-oriented architectures. In consequence, and as a direct application of such paradigm, Virtual Organization shared computational resources are virtualized by services. Nevertheless, the whole Virtual Organization cannot be characterized solely as the sum of the descriptions of these services; it must be described as a whole. And so far we have only characterized the functionality of isolated services.

In this section we propose several submodels each of them describing different facets of a Virtual Organization. The Virtual Organization Model of the Firenze Framework distinguishes...
between the *Agents Submodel*, which depicts the entities that may interact with the Virtual Organization (section 4.4.1 Agents Submodel); *Roles Submodel*, which is a representation of the patterns of behaviour that agents and services may adopt in the environment (section 4.4.2 Roles Submodel); *Events Submodel*, that contains a formal representation of the happenings of interest inside the Virtual Organization (section 4.4.3 Events Submodel); *Choreographies Submodel*, which depicts the global interactions among the constituent of the Virtual Organization (section 4.4.4 Choreography Submodel); *Themes Submodel*, which is a formal description of shared-functionality scattered through the Virtual Organization (section 4.4.5 Themes Submodel); and finally, *Organizations Submodel*, which depicts the different organizations involved in the institution that the Virtual Organization embodies (section 4.4.6 Organizations Submodel).

### 4.4.1 Agents Submodel

As an *agent* we refer to any human or software system able to execute a task in a certain domain (Schreiber et al., 1999). Newell’s definition of the Knowledge Level was agent-centric. Systems were modelled from an external point of view as agents. Newell’s intention was to rationalise their behaviour as they interacted with their environment. An agent possessed a highly unstructured nature, divided in a body of knowledge about its surrounding environment and a set of goals that motivated its behaviour according to the principle of rationality. Nonetheless, as the Firenze Framework aim is to model knowledge, not to predict behaviour, and therefore we propose a clear structure on knowledge (similar to the structures which were referred as a KL-models in (Van de Velde, 1993)). In accordance, agents descriptions are also highly structured, structure that is described by this submodel.

---

9 *If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action* (Newell, 1982)
The Agents Submodel formalizes the characteristics and capabilities of those individuals that may interact with (or within) the Virtual Organization; and it also sets its authorization and entitlement policies. Note that agents are not necessarily entities external to the Virtual Organization. Since their definition includes any software system, even internal modules of the Virtual Organization can be modelled as agents (e.g., an enactor for submitted jobs can be considered an agent, even though it is internal to the institution). The actions an agent may carry out to interact with the environment correspond to service operations formally depicted as configured Problem-Solving Methods, as shown in Figure 4-40. These actions usually represent interaction with the real world, which in the case of a Virtual Organization represents an interaction with its constituent computational resources.

We divide the agents model in two main conceptualization bodies, the Agents Taxonomy that enumerates the types of agents present in the environment; and the Agent Privileges Specification, which bounds the agent classes with the actions that they might perform in the environment (i.e. the Virtual Organization). More exactly:

- **Agents Taxonomy.** Each Virtual Organization has associated a set of agent ontologies. The concepts of these ontologies represent the different types of agents that may exist in the domain of the Virtual Organization. These concepts are linked using subsumption relationships. Therefore this tree-shaped structure (the root is the Agent concept) contains all the possible types of agents that may interact with or within the Virtual Organization and their inclusion associations.
Firenze Framework

Figure 4-41 Agent description main concepts and relationships.

- **Agents Privileges Specification.** The Agents Privileges Specification outlines the nature of the activities that an agent is allowed to carry out in the Virtual Organization. This model is comprised by the agent protection domains definitions, the set of different constraints that form a dynamic model of authorization; and the agent entrustment definitions, a formal description of the possibility of the delegation of privileges between agents. More precisely:

  - **Agents Protection Domain Definition.** The Agents Protection Domain Definition associates each of the agent types that belong to the agent ontologies to at least one protection domain (Saltzer and Schroeder, 1975). A protection domain in the context of this work is viewed as the set of resources that may be directly accessible for an agent; and the actions that it may perform on them. Protection domains are sculpted using a set of constraints, which are applicable to different facets of the formal descriptions to the services that virtualize the resources. More accurately these constraints are:

    - **Tasks constraints.** These constraints filter the set of tasks that the agent may perform. In consequence, tasks constraints formalize what the agent may carry out in the Virtual Organization.

    - **Method constraints.** These constraints sieve the set of methods (either primitive or complex methods) that the agent may accomplish in the Virtual Organization. Therefore, the method constraints formalize how the agent may carry out a task that it is allowed to do.
- **Domain constraints.** These constraints define the domain where the agent may operate. Agent actions are formalized using configured Problem-Solving Methods, as they are attached to a concrete domain of operation.

- **Non-Functional constraints.** Non-functional properties can also be taken into consideration in order to characterize the actions that an agent might accomplish. As we described in section 4.3.1.2 Non-Functional Profile, those properties are advertised in a formal manner in the service non-functional profile (see Figure 4-26). Therefore, the non-functional constraints range over the properties contained in this profile specifying their acceptable values.

![Figure 4-42 Constraints specification order.](image)

In Figure 4-42 we describe the order in which the constraints should be specified. First of all at least one task constraint must be specified, in order to delimitate the nature of the operations that the agent may carry out. Then we may define either a set of method constraints and/or non-functional constraints. At last we may express some additional restrictions on the domain.

- **Agent Entrustment Definition.** The Agent Entrustment Definition formalizes situations where an agent may delegate some of its privileges to other agent; and the circumstances that make this handing over possible. The main elements of this model are entrustments (represented as the **Entrustment** concept in Figure 4-41). The agents involved in the entrustment are related by means of **entrustor** and **entrustee** term relationships. The set of rights that are involved in the entrustment are represented as a group of access privileges formalized as **Access Constraint** instances, which are linked to the entrustment using the **consistsOf** relationship. Finally, depending on its nature, entrustments may be either unreserved either conditional. In consequence, we may optionally relate an entrustment with a condition that expresses under what circumstances the entrustment is effective. This condition is formalized using an **Expression** instance related with the entrustment with the **dependsOn** term relation.
4.4.2 Roles Submodel

The Roles Submodel defines and relates roles according to their taxonomical relationships and the way they interact. Roles are defined in this framework both as an abstraction mechanism from agent individuals; and as a mean to categorize services that belong to the Virtual Organization. This categorization is made on terms of their characteristics (e.g. non-functional and functional properties, model of interaction, resources that they use, etc.). Thus, our view of a role is twofold. From a service perspective they become classes of services that share common characteristics. Nonetheless, from an agent perspective, we consider roles as standardized patterns of behaviour. More precisely:

- **Roles provide an abstraction and an indirection layer over a collection of services that belong to the Virtual Organization.** On the one hand, a role can be used as a single entry-point for a class of services (in a similar manner to JINI (JINI, 2001) or JXTA services (JXTA, 2007). Roles can be considered a variable that contains an unspecified service of that class of services.

- **Roles provide patterns of behaviour to external agents in order to interact with the Virtual Organization.** This facet of roles is similar to the one presented in (Esteva et al., 2001) (Sierra et al., 2004)) for the description of Electronic Institutions. As we will spot later in this section (and we will be further described afterwards in the section of the choreography model, section 4.4.4 Choreography Submodel), in order to interact with the Virtual Organization agents should play a role in the scenes that depict Virtual Organization global interactions.

Figure 4-43 represents a general view of how the roles model fits in the overall Virtual Organization picture. Roles are used in the definition of interaction scenarios between services and agents produced in the system, what we refer as scenes (we describe this concept later in section 4.4.4 Choreography Submodel). In Figure 4-43 a simple scene is shown, scene0. Scenes are defined using roles in order to detach them from concrete services and agents. For example agenti, in order to participate in the global interaction depicted scene0 plays roleA. Services also participate in the scene, abstracted also by means of roles. In this concrete example roleB and roleC.
We decompose the Roles Submodel in three main parts, namely the Roles Taxonomy, which enumerates and relates the different kind of roles that are considered in the Virtual Organization; the Roles Constraints Specification that defines the set of constraints that characterize each role; and finally the Roles Events Specification describes the collection of asynchronous events that the entity playing a concrete role is able to produce/consume in the environment.

- **Roles Taxonomies.** For each Virtual Organization we identify a set of role taxonomies. Its concepts are the different types of roles that coexist in the Virtual Organization. Its relationships can be either domain-dependant or taxonomic. More specifically:
  
  - **Domain-dependent relationships.** In Roles Taxonomies ad-hoc relationships can be declared, in the same manner as they can be defined in any other ontology.
  
  - **Taxonomic relationships.** We define a small set of taxonomic relationships used to organize roles in a domain-independent manner. We identify the following relations for creating taxonomies:

![Diagram](image)

Figure 4-44 Taxonomic relationships of the roles ontologies.
- **subRoleOf relationship.** This type of taxonomical relationship allows the specialization of roles. The result role is the union of all its super-classes. It inherits their attributes, and more importantly, all the access constraints of its super-roles. Figure 4-44.a shows the graphical representation of this relation.

- **disjointRoleFrom relationship.** Used to assert that playing certain role in the Virtual Organization is mutually exclusive to performing other role. As shown in (Figure 4-44.b) we can define several roles as being disjoint.

- **intersectionRoleOf relationship.** This relationship permits the definition of a role as the intersection of a group of other roles (see Figure 4-44.c). The resulting role can be characterized as the maximum common denominator of the set of roles.

- **Roles Events Specification.** The Roles Events Specification intends to make unequivocal the different asynchronous events that an agent may consume or produce on the environment. In this context, we are only concerned about the events that belong to the Virtual Organization Events Submodel (this model will be thoroughly in the next section, 4.4.3 Events Submodel).

- **Roles Constraints Specification.** Roles represent a collection of services that share some common characteristics, as we have stated. We formalize these common characteristics using a set of constraints; those services that do not accomplish these constraints can not play this role. These constraints may range over the different aspects of a service and agent description.

![Role description main concepts and relationships.](image-url)
• **Functional constraints.** This kind of constraints restricts the tasks and operations that a service should have. A service should have certain operations in order to belong to a role.

• **Non-functional constraints.** They impose restrictions on the set of values for non-functional properties that membership to a concrete role imposes.

• **Interaction Process Constraints.** A role may impose certain message interchange compliance just by defining an interaction process and a type of relationship that the interaction process of the service should have (types might be bisimulation, which states that two processes behave in the same fashion, see (Milner, 1990); and equivalence).

### 4.4.3 Events Submodel

An event signifies an activity (Luckham, 2002); it is a record of an activity in a system. They can have several natures (we may consider temporal events, physical events, etc.), but in the context of this work we consider an event as an arbitrary detectable state change in a computer system (Mühl et al., 2006). This computer system, as we focus our attention on Grid environments, belongs or may be of some interest to the Virtual Organization. An event may have several physical incarnations, which we refer as notifications. A notification is a datum that reifies an event (Mühl et al., 2006); it contains data describing a concrete realization of a type of events.

![Figure 4-46 Event description main concepts and relationships.](image)

Events may be simple or composite (represented by *SimpleEvent* and *CompositeEvent* of the events model, as shown in Figure 4-46). *Simple events* are atomic, a single instant occurs between its beginning and its end, making them indistinguishable from the observer point of view. *Composite events* are the aggregation of other events, either simple or composite.
Composite events are described using an event expression that formalizes them (using the EventExpression and the definedBy relationship). Those events can be widely dispersed in time; and different relationships among them may exist (as we will describe in the following sections). The rationale for considering the existence of composite events is the possibility of defining event abstraction hierarchies; the capacity of defining fine-grained event subscriptions; and the possibility of outlining of the concept of global state. Let us discuss these ideas more thoroughly:

- **Event abstraction hierarchies.** Events can be arranged from low-level events to high-level events, what we will refer as event abstraction hierarchy. Figure 4-47 shows a schematic representation of how these hierarchies are defined. The composite event definitions constitute a set of event aggregation rules for each level. As an example, let us suppose that there is a composite event type event\(_a\) defined as a sequence of events (described by means of the event expression\(_a\) shown in Figure 4-47). Once in the level\(_i\), if an expression\(_a\) occurs, it can be inferred that in level\(_{i+1}\) the event\(_a\) occurs. The event abstraction hierarchy mechanism is vital for composite event systems, as it provides two basic features:

  - **Eventflow abstraction.** Higher-level events are constructed from the observation of events of the hierarchy lower layers. It can be used to filter insignificant events, but more importantly, to detect hidden situations of interest that emanate from the actual event interchange produced in the system.

![Figure 4-47 Events abstraction hierarchy.](image)
**Personalized context-aware system views.** Different alternative perspectives of the eventflow can be defined by defining different abstraction hierarchies (i.e. defining different composite events from similar observations). As shown in (Luckham, 2002) different event abstraction hierarchies can be used to specify and implement personalized views of a target system for each stakeholder in the system.

- *Finer-grained subscription.* As exposed in (Mühl et al., 2006) subscribers are usually only interested in few events, and may be overwhelmed by the huge number of simple event that may occur in the system. The use of just basic types of event may be too coarse, and it may lead to a notification flooding. The solution to this problem can be approached from an implementation level perspective. The addition of extra client logic in each of the subscribers (in the form of a distributed filters as shown in (Hohpe and Woolf, 2003)) is an option, but it will consume unnecessary resources, as events will flood the system. Our option is to consider the middleware as responsible of routing notification in more resource-safe way. In any case, a formal and explicit description of the interests of concrete subscriber is necessary in order to forward just the events they are interested.

- *Global distributed state.* Event-based environments are by nature highly distributed and decoupled. As presented in (Vera et al., 1999) in such systems the notion of global state is difficult to grasp. Composite events are effective for representing environment states in a very compact manner (see (Romer and Mattern, 2004)), since they do not picture the whole state of the system, but focus on significant state changes in relevant intervals of time.

![Figure 4-48 Events Submodel overview.](image-url)
Figure 4-48 represents a general view of how the events model is related with entities defined in the other submodels contained in the Virtual Organization description. The Events Submodel is closely related with the Agents Submodel and the Roles Submodel. As we have already stated, in order to interact with the Virtual Organization both agents and services play roles. As shown in the Roles Submodel, the definition of those roles includes the description of the event notifications that they are interested in; and the events that the roles may produce in the eventflow of the system.

The Events Submodel contains both the definitions and the relationships among the relevant events observable in the Virtual Organization. That includes the definition of an events taxonomy that enumerates and represents the hierarchy of the different kind of events that may happen in the Virtual Organization; and the association of composite events the high-level aggregation expressions that define them. We include all this information in both the Events Ontology and the Composite Events Specification.

- **Events Taxonomy.** For each Virtual Organization we define an event taxonomy. Its concepts are the different types of events (both simple and composite) that may occur inside the environment eventflow. These concepts will be linked using subsumption relationships, becoming the root element the Event concept.

- **Composite Events Specification.** Composite events, as we have already affirmed, are aggregations of other simple or composite events. The Composite Events Specification associates each of the composite events contained inside the events ontologies with an event expression that decomposes them. The aforementioned relationships provide the necessary mechanism to define different event abstraction layers in the system, and event expressions are used to describe composite events. The requisites that we identify for these composite events description mechanisms are the following:
  
  - **Expressivity.** Composite events descriptions must be expressive enough to be able to describe common composite event specifications. It must provide relationships between events (logical relationships, timing, restrictions over the attributes of events, etc.).
  
  - **Efficiency and scalability.** Though we model the flow of patterns using an abstract representation, we must bear in mind that the matchmaking of complex events has a strong efficiency and scalability requirements. As seems obvious, there are a trade-off between this requirement and expressivity.
• **Level of description.** As the rest of the framework, the description of events should be made as close to the knowledge level as possible. We should provide a representation independent description of the eventflow, and make the description of the eventflow consistent with the rest of the description presented in this work. It dethatches the description of event expressions from to any representation formalism and more importantly, from any event detection mechanisms (they will be presented in the next chapter devoted to the architecture, Chapter 5 Vega).

In order to cope with these requirements we propose the use of several eventflow patterns for building event expressions, as we have used to define the orchestration of methods and the interaction with services. We will describe them in the following section.

### 4.4.3.1 Eventflow Patterns

In this section we illustrate the catalogue of patterns that we identify for describing composite events (we present them in Figure 4-49).

![Figure 4-49 Eventflow patterns catalogue.](image)

We distinguish *property eventflow patterns*, which describe situations that concern the attributes of event; *structural eventflow patterns* that describe causal relationship among events; and *logic eventflow patterns* that describe situations where are a complex happening can better depicted as set of logically related simpler events.
Property eventflow patterns represent situations where certain criteria or relationship on the properties can be observed in the set of events of the eventflow. They are used as basic building blocks to describe more complicated event expressions.

4.4.3.1.1 Property Discriminator Eventflow Pattern

**Description:** A subset of the events in the eventflow exhibits some common characteristic in their attributes. In case that this discrimination criteria is not specified, we consider that this pattern describes the events that belong to that class (i.e. all the notification of the type of event present in the pattern).

**Graphical representation:** Figure 4-50 depicts how the property discriminator eventflow pattern is graphically represented. The constraint is represented as the triplet (attr, op, val) where attr is the attribute of the event, op is a Boolean operator, and val is the value that the constraint specifies.

![Figure 4-50 Property discriminator eventflow pattern graphical representation.](image)

**Synonyms or related approaches:** Rapide-EPL (Luckham, 2002) allows the definition of constraints over attributes of events by means of guards, which are defined using the where clause.

4.4.3.1.1.2 Property Relater Eventflow Pattern

**Description:** An attribute (or set of attributes) of an event in the eventflow (either simple or composite) is by some means related with an attribute (or set of attributes) of other event in the eventflow (which also can be simple or composite).

**Graphical representation:** Figure 4-51 shows how the property relater eventflow pattern is graphically represented. We relate the attributes using a set of relationships that we label in the picture as the relater.
Synonyms or related approaches: Rapide-EPL (Luckham, 2002) allows definition of similar situations using logic programming alike mechanisms (i.e. shared variables and substitution mechanisms).

4.4.3.1.2 Structural Eventflow Patterns

Structural eventflow patterns represent the core of the description of complex patterns, as they represent situations where certain notification sequences appear in the eventflow. We distinguish three structural eventflow patterns, the causality eventflow pattern, the iteration eventflow pattern, and the sequence eventflow pattern.

4.4.3.1.2.1 Causality Eventflow Pattern

Description: Causality is a recurrent situation in the eventflow where the appearance of an event (simple or composite) always produces the occurrence of other events (simple or composite). We denote the first event endorsed in this relationship as cause event; and we refer as later events as the consequence events.

In this type of relationship between a cause and effect event relationship, it is important to consider that the Cause-Time Axiom holds. According to (Luckham, 2002), the Cause-Time Axiom estates that if an event $event_a$ caused the $event_b$ in a system, then no clock in the system gives $event_b$ an earlier timestamp than it gives $event_a$ (Luckham, 2002). In other words, if $event_a$ causes $event_b$, $event_a$ must happen before. As we have defined simple event as atomic, in the case of a sequence of two simple events time overlap is impossible by definition unless both events occur at the very same time; therefore Cause-Time Axiom entitles that the cause event timestamp must be previous to the timestamp of the consequence event. However, when composite events come to scene, the situation changes. As shown in Figure 4-52, composite events timestamp is expressed as an interval that we denote event timeline. The timeline of a composite event is noted as an interval $[T_{lower}, T_{upper}]$. $T_{lower}$ is the lower bound timestamp of the set of events that compose the composite event, whereas $T_{upper}$ is the upper bound of the
timestamps of the set of event that compose the composite event (see composite event $A$ in the Figure 4-52).

As we consider causality a strict timing relationship, that means that the $T_{\text{upper}}$ of the cause event must be lower than the $T_{\text{lower}}$ of the effect event. We represent this situation in Figure 4-52, if composite event $A$ causes composite event $B$, necessarily $t^A_{\text{lower}}$ must be greater than $t^B_{\text{upper}}$. The elapsed time between the previous event and the later event can be specified optionally; in such case the difference between these timestamps equals the specified time. Finally, in the case of a mixed scenario, where both simple and composite events are involved, the situation is conceptually the same as in the case of two composite events, being $T_{\text{lower}} = T_{\text{upper}}$ in the case of the simple event.

**Graphical representation:** Figure 4-53 shows how the causality pattern is graphically represented. An arc, headed from the cause event to each effect event, is created to represent the direction of the relationship. In the arc a parameter *time* can be optionally specified.

**Synonyms or related approaches:** This pattern semantics are similar to the Rapide-EPL causality operator (Luckham, 2002).
4.4.3.1.2.2 Iteration Eventflow Pattern

**Description:** In the eventflow certain event (either simple or composite) repeats over time. Optionally, the number of iterations can be specified.

**Graphical description:** Figure 4-54 shows the graphical representation of the iteration pattern. The event expression of interest is surrounded with a doted square. An asterisk is added to the top-right corner to represent the iteration if we do not specify the number of iterations (see Figure 4-54.a). If we specify the number of iterations, the number is added to the top-right corner (see Figure 4-54.b).

![Figure 4-54 Iteration eventflow pattern graphical representation.](image)

**Synonyms or related approaches:** Rapide-EPL provides mechanisms for defining repetitive event structures, the \( \ll \) operator.

4.4.3.1.2.3 Sequence Eventflow Pattern

**Description:** An order in a pair of events can be observed in the eventflow. An event (previous event in Figure 4-55) is followed by other event (later event in Figure 4-55). Both later and previous events can be simple or composite. We take into consideration just strict sequences; the previous event cannot overlap with the later event and is possible that both events coincide in time. As in the case of 4.4.3.1.2.1 Causality Eventflow Pattern we have the same issue of the duration of composite events in such cases. The same conditions described there apply in the case of this pattern. The minimum elapsed time between the previous event and the later event can be specified optionally.

**Graphical Representation:** In Figure 4-55 we represent the sequence eventflow pattern. The optional parameter `time` specifies the time that must last between the events.
Synonyms or related approaches: The situations portrayed by this pattern without the optional time parameter can be formalized by the exist sequence operator (both in WSCDL and WSCI) and the commonKADS SEQUENCE operator (represented with the ; symbol) has also the same meaning. The time condition can obtained combining other language constructs (e.g. WSCI delay).

4.4.3.1.3 Logical Eventflow Patterns

This type of patterns describe situations where a complex happening is depicted as set of related simpler events that either happen at the same time, alternatively; or when an event is better characterized as the absence of other.

4.4.3.1.3.1 Conjunction/ Disjunction Eventflow Patterns

Description: There is a subset of the events in the eventflow that can be better described as:

- Several events happening at the same time\textsuperscript{10}, which we refer as the conjunction eventflow pattern.
- Several alternative (but not necessarily disjunctive) events happening at the same time, which we refer as disjunction eventflow pattern.

This pattern is useful to build composite event expressions by combining simpler ones.

Graphical representation: Depending on the logic relationship among the events identified in the eventflow, we define two graphical representations. Figure 4-56.a correspond with the conjunction eventflow pattern and Figure 4-56.b corresponds to the disjunction eventflow pattern.

\textsuperscript{10} The notion of happening at the same time is just conceptual in a distributed system. Properly speaking it should be referred as “in an small interval of time”.

Figure 4-55 Sequence eventflow pattern graphical representation.
Synonyms or related approaches: The semantics of the conjunction/disjunction patterns are equal to those semantics of the Rapide-EL and/or operator (Luckham, 2002). The disjunction eventflow pattern is also equal to the Snoop $V$ operator (Chakravarthy and Mishra, 1994).

4.4.3.1.3.2 Negation Eventflow Pattern

Description: There are situations in which events of interest in the eventflow are those that do not match certain criteria.

Graphical representation: Figure 4-57 shows the graphical representation of the negation eventflow pattern. The event expression of interest is surrounded with a doted square, and a circle with the word NOT inside is added to represent the negation.

Synonyms or related approaches: The semantics of the negation eventflow pattern are partially similar to the semantics of the not operator in the Rapide-EPL language (Luckham, 2002). The notable difference is the arity, accounting that the Rapide-EPL not operator is binary. If we state $P$ not $Q$, that means that the set of matching events should match the expression $P$ but not the expression $Q$.

4.4.4 Choreography Submodel

According to (WS-Glossary, 2004), the choreography defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state. This exchange of messages between participants enables information communication and possibly the synchronization of their external state (WSCDL, 2005). In (Barros et al., 2007), this concept of choreography is further extended and choreographies are also meant to capture the interactions in which a collection of services
engage, but from the perspective on an ideal observer who would be able to see every interaction occurring between the involved services (and agents). This notion of external observer (“birds-eye” view according to (Barros et al., 2007)) is the exactly the same that Newell expressed as the view point from which knowledge level definitions should be made, and therefore this approach is the one that we adopt in this framework.

So far we have provided a description of how a service individually. The Choreography Submodel contains the components necessary for the description of the overall system choreography, describing how agents and services collaborate playing different roles inside the Virtual Organization. We formalize this choreography as a set of scenes. A scene is defined as the composition of multiple, well-separated and possibly concurrent interactions between multiple agents and services (concept based in the one described in (Esteva et al., 2001)(Sierra et al., 2004) for Electronic Institutions). As we have already stated, in order to abstract from concrete types of individuals (both agents and services), the collaborating parties in scenes are represented by means of roles (which belong to the afore described roles model). That means that in order to participate in a scene is mandatory that agents and services play at least one of these roles. Scenes thus provide the top-level view of the overall system collaboration; outlining the interactions among roles, and situating individual services and agents inside these roles.

Figure 4-58 Scene description main concepts and relationships.

A scene can be seen as a directed graph. Nodes represent the roles implicated in the scene (instances of the concept Role, related with an instance of either an Agent or a Service by means of the plays and belongsTo relationships respectively); whereas arches represent interactions among them (instances of the concept Interaction). It is important to remark that scenes definitions contain roles; the entities that play each of the roles remain unspecified (i.e. we can have several agents that might play a role; or multiple services that might belong to that role). Moreover, the interactions that appear among roles are neither ordered neither related. No further information about the exchange of messages that those interactions produce is provided, since all this information is settled on the interaction processes of the services that belong to
each of the scene roles (interaction processes that must obey the restrictions that are part of the definition of the role, see 4.4.2 Roles Submodel).

Let us summarize all these ideas in a graphical fashion in Figure 4-59. In the definition of sceneA we state that, in the concrete scenario that sceneA formalizes, multiples roles are involved (namely roleA, roleB, roleC, roleD); and that they collaborate by means of the interactions interactioni, interactionj, and interactionk. More precisely, roleA collaborates with roleB and roleC; and roleC with roleD; and these collaborations are embodied by the interactions interactioni, and interactionj; and interactionk respectively. The precise description of the order and the circumstances under which this interactions occur, are described at a lower level of abstraction in the interaction process of each of the involved services, and therefore are part of the Service Model. Going back to Figure 4-59 example, let say that serviceA plays roleA in the sceneA. In the interaction process of serviceA, the order of the interactions interactioni and interactionj is specified, and moreover, the precise message exchange of each interaction is provided. Interaction processes are descriptions made from an external observer perspective, but unlike scenes, they are focused on a single service. In this example, the interaction process of serviceA states that serviceA first carries out the interactioni, and then the interactionj.
4.4.5 Themes Submodel

*Themes Submodel* main endeavour is to capture and specify functionality shared among the services that are part of the Virtual Organization. This functionality is provided by the Virtual Organization itself, you can think of it as an infrastructure or commodity provided to services that belong to the system.

For defining this common functionality, we reuse and extend notions of the Aspect-Oriented Programming (sometimes referred as AOP) paradigm (Gregor et al., 1997); especially the approach described in (Clarke and Baniassad, 2005)(Harrison et al., 2002), where this paradigm is included in both design and analysis levels. The Aspect-Oriented Paradigm allows the definition of crosscutting portions of a system separately from its structural entities; providing thus a solution to the scattering of functionality. Common functionality shared among different services can be grouped and defined in a single point, extending the coherency and manageability of the overall system.

![Themes taxonomy](image)

Figure 4-60 Themes taxonomy.

The basic entities of the Themes Submodel are themes. A *theme* is an encapsulation of a concern, being a *concern* any kind of functionality in the system (either a feature or some type of processing). Depending on the nature of these themes, and following the classification proposed in (Clarke and Baniassad, 2005), we categorize them in two main groups: asymmetric themes (we refer to them as aspects, see Figure 4-60) and symmetric themes (or facets, as shown in Figure 4-60). We specify them separately in two separate specifications.

- **Asymmetric Themes Specification.** The Asymmetric Themes Specification defines the elements necessary to describe functionality horizontal to the whole Virtual Organization, and it is composed by aspects. *Aspects* encapsulate a set of advices, which are pieces of behaviour that are triggered at certain strategic points in the execution of services operations. An advice is shared along the Virtual Organization, and is triggered in multiple situations under a well-defined set of observable circumstances. As we are considering service-oriented Virtual Organizations, the functionality of advices is provided transparently by other service.
Figure 4-61 shows the main concepts considered in the specification of aspects. Firstly, aspects can be considered aggregations of advices. In consequence, an instance of the concept Aspect may be related a set of instances of Advice (at least one). The concept of Advice represents the triggered behaviour; behaviour that as we have discussed is provided by another service operation. To formalize such relationship we associate each advice with one (and only one) operation (relation realizedBy with Operation).

A joinpoint is a possible execution point for the advice. As we consider the set of service operations as valid candidates, we establish a hasJoinpoint relation with the Operation concept (we restrict to just one joinpoint operation per advice).

Pointcuts are predicates that determine, once the execution reaches the joinpoint operation (i.e. once joinpoint is going to be executed), whether its related advice should be triggered or not. This is formalized as illustrated in Figure 4-61 by associating the concept Advice with an Expression by means of the hasJoinpoint relationship. The evaluation of this pointcut expression determines whether the advice should be triggered or not.

Once we reach a pointcut operation, and the joinpoint expression is matched, the type of the advice determines how the advice is triggered (i.e. the order of the execution of the advised operation and the actual operation to be executed). We distinguish three types of advice, namely BeforeAdvice, AfterAdvice, and AroundAdvice (as shown in Figure 4-61). In the case of the BeforeAdvice, the operation related with the advice is executed before the matching joinpoint operation is invoked; when it is an AfterAdvice, once the operation has been successfully carried out, the advice joinpoint operation is executed; and finally in the case of AroundAdvice, the advice joinpoint operation is executed instead of the service
Firenze Framework

operation. How the inputs/outputs of the operation parameters and the weaved advise are related is defined by means of explicit mappings.

- **Symmetric Themes Specification.** The Symmetric Themes Specification defines the elements that we use to specify common functionality orthogonal to the Virtual Organization services functionality. This orthogonal functionality is grouped in facets, which encapsulate common and directly invocable concerns that are orthogonal to the core functionality of the Virtual Organization services.

![Diagram](image)

Figure 4-62 Facet descriptions main concepts and relationships.

As shown in Figure 4-62 facets are composed by operation providers, services that group common and directly invocable functionality. These operation providers are attached to services, to which they provide new operations (attachedTo relationship); and they are implemented by services (realizedBy relationship). The interaction process of the service that has been complemented with an operation provider is the combination of both the interaction processes of the service and the one that realizes the operation provider running in parallel. We depict this idea in Figure 4-63. A service that exposes N operations (operation\(_1\)...operation\(_N\)) is complemented with the operation provider operationProvider\(_i\). operationProvider\(_i\) is implemented by service\(_i\) that exposes M operations (operation\(_{N+1}\)...operation\(_{N+M}\)). From an internal perspective, the service remains unaltered; but from an external perspective, the service exposes the sum of these operations that can be invoked in parallel to the set of operations that the service already possessed.
4.4.6 Organizations Submodel

The Organizations Submodel depicts the “real” organizations that compose and are behind of the Virtual Organization. We consider an organization an entity that is established to accomplish some goals; and that in consequence provides computational resources and services (Parashar and Browne, 2005).

We specify an organization decomposing it in terms of its descriptive information, the set of services and virtualized computing resources that it shares, the hierarchy of roles, and the organization access policies.

- **Descriptive information.** The descriptive information is a set of non-functional properties (namely name, description, phone, fax, email, webURL) and a location (described using the Location concept (it is described using the same ontology that we proposed for services location availability in section 4.3.1.2 Non-Functional Profile).
Firenze Framework

- **Organization components.** Organizations share and expose services and resources. We establish relationships between an organization and its services (the provides relation shown in Figure 4-64) and the shared resources (the posses relation shown in Figure 4-64). Finally, the organization is associated with its providing policies (by means of the hasPolicy relationships, see Figure 4-64).

- **Organizational roles taxonomy.** It formalizes the classes of individuals inside the organization. Organizational roles provide a way to specify service entitlements to individuals within the organization; they are used in the Policy Specification to abstract access policies from concrete individuals. There can be several organizational roles ontologies. The concepts of these ontologies represent the types of organizational roles defined in the organization, including each concept related attributes. These concepts are related using the same relationships that we defined in 4.4.2 Roles Submodel for the Virtual Organization roles.

- **Policies Specification.** Each organizational unit must define its provisioning policies. These policies can be considered more than simple role-based access control specification, since depending on the nature of attributes they constraint, they may specify issues about security, privacy, authentication, traffic control etc. We adapt and conceptualize work the presented in WS-Policy (WS-Policy, 2007), XACML (XACML, 2005) (and more recently WS-XACML (WS-XACML, 2006)) to define a general purpose policy assertion model.

![Figure 4-65 Policy Submodel main concepts and relationships.](image-url)
A policy in the proposed model is defined as a set of assertions that describe a guiding principle for access granting. Thus, it is especially important to formalize policies so that external agents may be able to interpret them and know in anticipation whether they meet the requirements set by the organization in order to interact with a service. The definition of policies is decomposed in the following elements:

- **Applicability condition.** Each policy embodies different facets of the provisioning guidelines of an organization. Given a requestor entity and the operation that the requestor desires to carry on certain service, we need to clarify which of these different facets is concerned. Thus, for each policy we define its target (associating each instance of a policy with a target using the applicability relationship as shown in Figure 4-65). The target of a policy evaluates each action carried out by requestors. The target expression evaluates, depending on that action and on the requestor entity attributes, if the requested action is susceptible of being evaluated by a policy or not. The requestor, as shown in Figure 4-65 is associated with the action by means of the requestor term relation that associates the concept Action with the concept Organizational Role. The action represents what the requesting subject wishes to carry out on the organization resource, which as we consider that resources are virtualized by services, represents the invocation of an operation of certain service (see Figure 4-65).

- **Set of assertions.** Policies in the proposed model are embodied by a set of assertions (using the describedBy relationship as shown in Figure 4-65). Each assertion represents a set of acceptable values for types and attributes of policy compliant entities. Assertions thus categorize the set of individuals that are allowed to carry out the set of actions targeted by their policy. We distinguish between simple or complex assertions. Simple assertions contain the expression that formalizes the types of roles and/or acceptable role attributes values. Complex assertions are built up by means of other assertions, either simple or complex. For describing the composition of policies we propose a set of high-level policy design patterns, as we did previously in the case of the description of compound events, complex interactions, etc.

### 4.4.6.1 Policy Patterns

As in the case of composite events descriptions we propose a three patterns to depict complex policies. The main purpose of policy patterns is to provide a high-level domain and language independent mechanism to combine different policy assertions; and to provide a graphical representation in order to symbolize them. In the following section we provide for each of the policy patterns a description of the pattern and its policy context, its graphical...
representation, and synonyms in related approaches (as we have stated we have considered WS-Policy, XACML and WS-XACML).

4.4.6.1.1 **Exclusive Alternatives Policy Pattern**

**Description:** A complex policy assertion specification can be portrayed as a set of one or more mutually exclusive policy alternatives. An agent action is admissible in case that one (and only one) of the policy assertion is satisfied. Those enumerated assertions may be simple or complex.

**Graphical representation:** Figure 4-66 portraits the graphical representation of the exclusive alternatives policy pattern. All the admissible alternatives ($\text{assertion}_j \ldots \text{assertion}_k$) are linked to a node that characterizes the pattern. This node is represented as a rounded triangle with a multiplication sign in the middle.

![Figure 4-66 Exclusive alternatives policy pattern graphical representation.](image)

**Synonyms or related approaches:** This pattern semantics are similar to those of the $\text{wsp:ExactlyOne}$ operator tag present in the WS-Policy specification. In XACML all the policy assertions should be separately defined and combined using the First Applicable Policy combining algorithm.

4.4.6.1.2 **Conjunction Policy Pattern**

**Description:** The accordance to a policy implies the accordance to a list of policy assertions. Thus an agent action is admissible if it simultaneously satisfies the collection of policy assertions. Those enumerated assertions may be simple or complex.

**Graphical representation:** Figure 4-67 portraits the graphical representation of the conjunction policy pattern. All the admissible alternatives ($\text{assertion}_j \ldots \text{assertion}_k$) are linked to a node that characterizes the pattern. This node is represented as a rounded triangle with a dot in the middle.
Synonyms or related approaches: This situations described by this pattern are the same as the one depicted by the wsp:All operator tag of the WS-Policy specification.

4.4.6.1.3 Disjunction Policy Pattern

Description: The accordance to a policy implies the accordance to at least one of the policy assertions. Thus an agent action is admissible if there is at least one satisfied assertion among the collection of policy assertions. Those assertions may be simple or complex.

Graphical representation: Figure 4-68 portraits the graphical representation of the disjunction policy pattern. All the admissible alternatives \((assertion_1 \ldots assertion_k)\) are linked to a node that characterizes the pattern. This node is represented as a rounded triangle with a summation sign in the middle.

Synonyms or related approaches: This pattern semantics are similar to those of the wsp:ExactlyOne operator tag present in the WS-Policy specification. In XACML all the policy assertions should be separately defined and combined using the Only One Applicable Policy combining algorithm.
4.5 **Firenze Framework Design and Annotation Process**

In this section we present how we conceive the process of creating a design using the Firenze Framework. To describe an entity using the Firenze Framework means to instantiating a set of abstract models and ontologies. Once created, this abstract high-level design can be downgraded to the symbolic level into different concrete implementations, depending on the particular needs of the designer.

Creating a Firenze Framework description does not just consist on annotating services. All the hidden knowledge that Grid systems already contain should not be neglected; it must be made formal and explicit. Grid systems already handle huge amounts of metadata information (e.g., embedded in the application code, in database schemas, in XML schemas, etc.) related with several topics (e.g. domain knowledge, sharing policies knowledge, process description knowledge, etc.). All this metadata has associated semantics; the problem resides in making metadata explicit and accessible, since most of this metadata information is completely hidden and embedded in the software infrastructure. In addition this already in use metadata is its informal nature, interpretable only by human; and moreover, this kind of information is complex domain dependent knowledge. A classical knowledge acquisition process should be performed, especially in cases where we need to develop the whole Virtual Organization description. As described in (Van de Velde, 1993) the use of a model in terms of knowledge is equivalent to knowledge acquisition; the model itself can be used as a frame for a knowledge acquisition process. Firenze Framework provides models that provide such guidance; to populate them in a systematic and sound way such models we propose the following methodologies:

- **CommonKADS.** In order to carry out the knowledge acquisition process. We propose the use of commonKADS (Schreiber et al., 1999). CommonKADS is the leading methodology to support structured knowledge engineering. Developed and validated by different companies and universities in the context of the European ESPRIT IT Programme, it has became de facto standard for knowledge analysis and knowledge-intensive system development. Furthermore, part of the knowledge components that we use is based on those of CommonKADS (e.g. Problem-Solving Methods), so we consider it a perfect fit.

- **METHONTOLOGY.** For the creation of the different domain dependant ontologies as our knowledge representation model is built on top of WebODE knowledge representation model (Arpírez et al., 2003). Thus, we consider the METHONTOLOGY (Fernández-López et al., 1999) methodology as the most coherent choice is the use of for building the domain dependent ontologies, since the knowledge model of WebODE is based on the internal
representations proposed in METHONTOLOGY. But apart from this opportunistic reason, METHONTOLOGY enables the construction of ontologies at the knowledge level, guiding the knowledge expert in how to carry out the ontology development process. Its roots are the main activities identified by the IEEE software development process (IEEE, 1996) and it has been proposed for ontology construction by institutions such as FIPA\textsuperscript{11} (the Foundation for Intelligent Physical Agents).

Once we have created the core of the Virtual Organization description using these methodologies, the aggregation of new services descriptions to the overall Virtual Organization description is much simpler, since it is a strongly guided annotation process. Services are annotated using as reference the previously obtained models.

After the knowledge level description of the services is created, already existing services and entities should be annotated. For relating the description with the entities we wish to annotate, we propose to generalize the concept of Semantic Bindings introduced in S-OGSA (Corcho et al., 2006). Semantic Bindings are the entities that come into existence to represent the association of a Grid Entity with one or more Knowledge Entities (where Knowledge Entities correspond with ontologies, rules, knowledge bases or even free text descriptions that encapsulate knowledge that can be shared (Corcho et al., 2006)). In analogous manner, in the Firenze Framework entities within an environment are linked with their description using knowledge bindings. Knowledge bindings are groups of language independent assertions that link the annotated entity (i.e. Web service; any entity in a Grid environment, such as Grid service, resources, a message, etc.) with a knowledge entity (i.e. a nameable entity whose purpose is to describe an entity at the knowledge level). In consequence, on the one hand we propose to decouple completely the semantic layer from the syntactic layer (as shown in Figure 4-69); and on the other hand it brings the possibility that a service may posses from zero to many semantic descriptions, descriptions that can even dynamically change.

\textsuperscript{11} http://www.fipa.org/specs/fipa00086/
We consider that we have described a Virtual Organization, and its constituent services, once the instances of each of the ontologies that describe has been created, and the different annotated entities have been related with its knowledge level description by means of knowledge bindings (see Figure 4-69). The next step is the automatic translation of all the knowledge level descriptions into the symbol level using representational languages so that external agents and software are able to understand and make effective use of it.

The advantage regarding decoupling and reusability of using a knowledge level framework arises again. As we depict in Figure 4-70, the description of any of the entities described in the Firenze Framework (e.g. service, role, agent, etc.) may have from zero to many associated symbol level representations ($representation_1$ to $representation_N$); and each of them may be even expressed in different representational languages and formalisms; and with different detail granularities.
Chapter 5   Vega

This chapter is devoted to the event-driven architecture, the second main outcome of this thesis. We will describe this distributed knowledge-based architecture able to achieve an internet-scale semantic services provisioning. Its main characteristics are its inherent distributed nature and the change of mode of interaction. A set of intelligent peer-to-peer brokers are in charge of handling the notification decentralized backbone bus, around which a set of semantically annotated event-driven services are situated.

Vega (Vega is an event-driven Grid architecture) is a service-oriented event-driven knowledge-aware Grid architecture. Such definition contains the main characteristic of the architecture that we are going to present in this chapter. It is a multifaceted architecture, and as such, it can be defined depicting each of these facets (which we portrait in Figure 5-1). More precisely:

- As a service-oriented Grid architecture, Vega is composed by a set of decoupled services that virtualize real world resources. As a service, we refer to a network-enabled entity that provides some capability (we represent them in Figure 5-1 as the S’s that become the joints between both the event-driven and the service-oriented architecture). This set of services, apart of being directly invoked by external agents, may react to/generate events on the environment in an asynchronous and autonomous manner.

I am an architect; they call me a butcher,
I am a pioneer; they call me primitive.
The Manic Street Preachers,
Faster

Figure 5-1 Event-driven architectures and service-oriented architectures relationship.
The event-driven architecture facet of Vega results in a complete change in the model of interaction with respect to other semantic Web services based architectures. The system becomes responsible of the dissemination of event notifications to the pertinent services inside the system, which in practice means to provide a suitable reaction to these events using the functionality provided by the services. Finally, the system is also responsible for detecting the happening of composite events, that might result of the spatial or temporal combination of simple events.

When we refer to Vega as a knowledge-aware architecture we mean that it makes use of knowledge expressed in formal terms as key enabler of its core functionality. The active use of knowledge is present on each of the basic activities that Vega performs.

As we described in the chapter devoted to the objectives of this work, our goal when designing this architecture is the definition of a truly distributed knowledge-based architecture that achieves:

- **Self-configurability and self-optimization.** Activities such as the publication of a new event-driven service, the spawning of new services instances, its possible removal, the propagation of event notification, etc. must be done by the system in an automatic fashion. These activities involve changes in the routing topology of the system that should be introduced transparently, and as depicted in (Kephart and Chess, 2003), these updates should be carried out in a consistent manner autonomously. Moreover, the architecture should be able to detect and handle situations where changes in its configuration might result in a better overall system performance.

- **Scalability.** Albeit we are dealing with a knowledge-intensive environment, the result architecture should be able to handle a huge number of annotated services and the great notification throughput that they generate.

## 5.1 Vega Architectural Principles

Before we present the architecture, we are going to present the principles that drove its design. In order to achieve the objectives and goals that we have identified for the architecture, we have defined a set of desirable design principles, in the following sections we thoroughly depict these principles grouped in the four paradigms that appear in Figure 5-2.
We start describing the *distributed knowledge system paradigm* principles illustrate how to use knowledge in a distributed yet effective fashion; then we depict the *service-oriented paradigm*, which are basic principles of service-orientation paradigm described in (Erl, 2007) but updated with the inclusion of formal and explicit semantics; after that we present the *event-driven paradigm* with the set of design principles that characterize this kind of systems that adopt the event-based interaction model; and finally *autonomic computing paradigm*, that encompasses the design principles that deal with the construction of self-managed systems.

### 5.1.1 Distributed Knowledge System Paradigm

The proposed architecture takes advantage of knowledge that the components that build up the environment expressed in formal terms. As described in (Schreiber et al., 1999), knowledge systems have several benefits, among which we can enumerate faster and better decision-making, increased productivity, etc.. The concern with knowledge systems is that they have been labelled as non-scalable and problematic in terms of performance. We consider that these problems are not inherent at all to knowledge systems, but have to do with the architectural decisions made whilst designing and implementing them. Most of current knowledge systems adopt the classic client/server architecture and a remote procedure call model of interaction. This architectural choice has resulted in monolithic brokers that store huge quantities of information on the one hand; and brokers that need huge processing power to handle that information on the other.

Our premise is that we cannot only create scalable knowledge systems, but that we should use knowledge as an effective medium to achieve scalability. In accordance, we define a design paradigm based on that premise, which we strictly follow while defining the architecture. Let us describe each of its design principles in the forthcoming bullets.

- **Knowledge-based system principle.** On the basis of the knowledge-based system definition that appears in (Motta, 1999), we consider a knowledge-based system as complex decision-making systems that use problem solving knowledge when making decisions under
uncertain conditions. In accordance, this principle proposes the use of knowledge in the core activities of the system, enhancing thus its basic operations. Systems should not just handle, store, or discover knowledge but should also use it effectively to achieve their goals.

- **Lack of complete knowledge principle.** This principle states that the agents that together compose the architecture should make their decisions on the basis of the partial view that they possess of the environment using its own knowledge base. Each of the participants in the system is only aware to a certain extent of the overall system situation; we cannot assume the existence of a global shared blackboard, nor an oracle.

- **Distributed decision-making principle.** As we have already stated, centralization becomes weakness when it comes to scalability and robustness. In the earlier principle we tackled with the centralization of knowledge, let us now deal with the distribution of its use. We have postulated the inclusion of knowledge in the core of decision-making process. This principle states that it should be done in a highly decoupled distributed fashion, each of the agents involved in the decision-making process should act in parallel, and in line with the lack of complete knowledge principle, coordinated yet using their own knowledge base and messages from other agents.

### 5.1.2 Service-Oriented Paradigm Principles

Service-oriented computing has emerged as a paradigm for enabling the flexible interconnection of autonomously developed and operated applications within and across organisational boundaries (Alonso et al., 2003).

The service-oriented paradigm captures a distinct approach to the analysis, design, and implementation to all types of service-oriented IT environments, introducing a set of principles that govern diverse communication, architecture, and processing logic facets. In our case, we consider service-oriented systems that make use of formal semantics, so we have adapted these principles accordingly. Best defined in (Erl, 2007) these design principles are: the standardized service contract principle; loose coupling principle; abstraction principle; reusability principle; autonomy principle; statelessness principle; discoverability principle and composeability principle. Let us describe them in detail, making explicit where and how meaning-based extensions are present.

- **Discoverability and standardized service contract principles.** In order to make the description of service capabilities understandable to any interested party, the properties of a service should be compliant with some design standard, the service contract. The service contract may include any information regarding the identification of the services (e.g. URI,
name, textual description); functional properties, such as the type of the input/output parameters, interaction model; and non-functional properties, such as QoS (Quality of Service), the location of the service, security constraints, etc. The discoverability principle can be considered complementary to this principle. The discoverability principle states that we should annotate services with metadata to enable their discovery by interested parties. Moreover, as we are proposing a semantically enhanced architecture, these annotations should be explicit and expressed in formal terms in order to make them machine interpretable.

The combination of these principles supports the interpretability of services, resulting in an increase in the predictability of the service behaviour. The ability to predict the future behaviour of a service is a key mechanism to achieve scalability, since it allows the evaluation of the necessary computational resources required to enact a specific service. This mechanism enables the intelligent provisioning of resources to prevent software resources running out.

- **Loose coupling principle.** The loose coupling principle states that the interface of a service should impose low consumer coupling and should also be orthogonal to its surrounding environment. Loose coupling, as presented in (Kayne, 2003), intentionally sacrifices precision in the description of the interfaces of services for a greater benefit: the achievement of flexible interoperability among systems that are heterogeneous with respect to technology, location, performance, and availability. Loosely coupled applications aim to be more reusable and adaptable to new requirements. Loosely coupled systems, such as our choice, event-driven systems (Luckham, 2002)(Mühl et al., 2006) have proven to be highly scalable when compared to traditional tightly coupled systems.

The problem with loosely coupled systems is the risk of creating too sparse systems, with little or no robustness (see (Erl, 2007)). The relation among different component of the architecture becomes too dim, what makes the system difficult to govern, makes it more inefficient, makes necessary the inclusion of additional logic in each of the components, and a long etcetera of undesired side effects. This is where a knowledge-intensive approach is very helpful; knowledge about the domain where the system is used, knowledge about the role of each of the components inside the system, knowledge about context, etc. can be used to raise the intelligence level of the middleware that glues the system. And this gluing can be done transparently without introducing unnecessary coupling among the environment components.
o **Abstraction principle.** The abstraction principle dictates that the details of software artefacts that are not indispensable for others to effectively use it should be hidden. All the information necessary to invoke the service is contained in the service contract; and all the knowledge of the underlying logic, technology, etc. should be completely buried. This principle can be considered as a synonym of the old software engineering concept of black boxing. The abstraction principle enables replaceability, which as outlined in (Armstrong, 2007), combined with fault isolation and fault recovery, enhances scalability.

o **Reusability principle.** The reusability principle states the functionality provided by services should be as domain and context independent as feasible, facilitating reuse (Erl, 2007). As a direct consequence of the application of this principle the logic of a service should be highly generic, independent from its original usage scenario. The reusability principle is a key enabler for service-oriented infrastructures, since it makes possible the creation of huge libraries of domain-independent services that leverage the construction of new complex context-dependent services.

o **Autonomy principle.** The autonomy principle states that services should be able to carry out their processes independently from outside influences. The only way to affect the results of a service should be through the modification of the input parameters as specified in the service contract. Service autonomy increases reliability and more importantly predictability and fault isolation, which as presented in (Armstrong, 2007) leads to an increase of the overall system scalability.

o **Statelessness principle.** The statelessness principle dictates that services should minimize resource consumption by deferring the management of state information when necessary (Erl, 2007). This notion of statelessness has been taken to the extreme by the REST architectural style (Fielding and Thomas, 2000), which has also been successfully applied to service-oriented architectures in recent years. Conformance with the statelessness brings many benefits (e.g. fault tolerance); but in the context of the presented work we consider this principle because is vital for the scalability of the entire service-oriented infrastructure. State maintenance is one of the most resource consuming tasks in a computer infrastructure. A reduction of the amount of state information taken into account by each service, albeit small, produces a significant reduction in the resources used by the overall system.

o **Composeability principle.** The composeability principle identifies services as effective composition participants, regardless of the size and complexity of the composition (Erl, 2007). From a bottom-up perspective, we consider combining simpler services into larger
services; from a top-down view, service composition is an effective way to tackle with the complexity of certain types of processes.

### 5.1.3 Event-driven Paradigm Principles

Let us now enunciate in an analogous manner to the previous sections the principles related with the event-driven facet of Vega. As described in (Cazaniga et al., 1998), the main novelty and innovation of the event-based architectural style is that neither the published notifications nor the subscriptions are directed toward specific components. The producer of the event notification is unaware of its consumers; the event-based system is the solely responsible of delivering these notifications to the consumer modules. The main characteristics of event-driven architectures are identified in (Luckham, 2002) (Mühl et al., 2006)(Hohpe, 2006), but they can be summarized stating that event-based systems carry the potential for easy integration of autonomous, heterogeneous components into complex systems that are easy to evolve and scale (Bates et al., 1998)(Mühl et al., 2006). We consider the principles that govern such systems the following:

- **Event-based interaction model principle.** In event-based systems the initiator of communication is the provider of data, that is, the producer of the event notification. Moreover, as presented in (Mühl et al., 2006), providers are not aware of the consumers, and more importantly, a notification provider must not publish a notification with the intention of triggering other a consumer activity in the system since (making certain assumptions about the competency of the consumer clearly would introduce functional coupling). Last, but not least, this model also dictates that communications are made in an asynchronous fashion. Agents do not wait for the receiving system to process events, instead they can continue carrying out other activities.

- **Efficient broadcast communications principle.** As we have stated, participating individuals broadcast events to any interested party. More than one component can listen to an event and process it, and more importantly, the generator of the event is not aware of which might be this processor *a priori*. Even though, the delivery of events should not be based upon a
general flooding of the event on system. Instead, the system should take the responsibility of delivering the notification as efficiently as possible\textsuperscript{12} to the interested peers.

- **Timeliness and fine granularity principles.** Agents publish notifications of events as soon as they are aware of their occurrence. Depending on the nature of the event, the event might be locally stored for a later processing cycle. Closely related with the timeliness principle, the granularity principle states that agents tend to publish a myriad of individual events as opposed to a single aggregated event.

- **Common model principle.** All the peers of the event-driven system share nomenclature among participating peers, which at least classifies events (e.g. some form of hierarchy). Even in “conventional” (i.e. with no semantic annotation) software engineering approaches like (Hohpe, 2006), the necessity of an ontology shared across the even-driven system is outlined. It allows components that may be plugged into the event system in the future to posses a common vocabulary to express their interest in an understandable and unequivocal manner.

- **Complex event processing principle.** The system (i.e. the event-driven architecture) understands and monitors the relationships between events (e.g. aggregation, causality, etc.). The client expresses its interest in a certain type of complex event, and expects to be notified when a notification of such event is produced in the system. That means that the logic necessary to detect complex events is not on the client side; it is part of the system responsibility.

### 5.1.4 Autonomic Computing Principles

The concept of Autonomic Computing was first introduced in (Kephart and Chess, 2003) where autonomous systems were characterized as self-manageable systems. In the context of services, the idea of being able to self-manage is very important, especially when we consider systems that are created by composing services offered by third parties. Such system does not have any control on the way the component services are actually offered. Therefore, they should be equipped with some self-management capability that would allow them to react to the cases

\textsuperscript{12}As we shall see, we have to be careful with this principle. The “delivering the notification as efficiently as possible” clause is not casual but intended, since in some situation the search of a greater goal, like scalability, may force us to sacrifice some efficiency in exchange. Nevertheless, this must be done in certain exceptional cases, and not as a usual policy. We describe this topic later on when we describe the expertise-based routing algorithm.
in which these services provide incorrect results or are unresponsive. As is identified in (Parashar and Hariri, 2006) the basic requisites for a self-manageable system are:

- **Knowledge aware.** The system should possess knowledge about all its components (capability, actual status, etc.); and also knowledge about the context of its activity and those of other resources within the infrastructure.

- **Able to sense and analyze environmental conditions.** This includes both the ability to proactively inspect the components of the environment and to introduce changes looking for ways to improve its functions.

- **Able to actuate on its environment.** The self-manageable system should be able to plan for and affect changes by altering its own state and effecting changes on its surrounding environment.

The characteristics of autonomous systems are being applied today in four fundamental areas of self-management to drive significant operational improvements where traditional manual-based processes are neither efficient nor effective. These four areas are related to different attributes of autonomous systems, and they are the capabilities of the system of self-configuring, self-healing, self-optimizing, and self-protecting. For each of these main areas of applicability, a design principle can be extracted and could be incorporated in the proposed architecture\(^\text{13}\). These principles leverage the construction, configuration and deployment of infrastructures that enable Web scale service-oriented environments.

- **Self-healing principle.** According to the self-healing principle, computing systems should be able to detect, diagnose, and repair localized problems resulting from failures both in software and hardware. With this purpose, systems should analyze the monitored information (ranging from simple log files to more structured and complex provenance information), detect possible problems, and propose and enact actions to tackle them.

- **Self-configuration principle.** The self-configuration principle states that systems should configure themselves automatically in accordance with high-level declarative policies. These policies specify what is desired, not how it is to be accomplished (Kephart and Chess, 2003). The deployment, configuration, and integration of large, complex or highly changing systems is a challenging, time-consuming, and error-prone task even for experts. When a component is introduced in the environment, it should announce its capacity and should be

\(^\text{13}\) We enumerate of all them for the sake of completeness. In the presented work we restrict ourselves two just the self-configuration and self-optimization facets of autonomic computing.
incorporated seamlessly. The rest of the system should then adapt to its presence and be aware of its functionality.

- **Self-optimization principle.** The system should continually seek ways to improve its operation, identifying and seizing opportunities to make themselves more efficient in performance or cost (Kephart and Chess, 2003). Once more, this principle implies that the system should be able to monitor itself and be able to carry out actions to tune its actions accordingly to the monitored information. As described by (Miller, 2005), these tuning actions could mean different things, such as resources reallocation, dynamic workloads management, replication of functional elements to prevent overloaded elements, and a long etcetera.

- **Self-protection principle.** The self-protection principle can be seen as two different but correlated facets, since it declares that the environment should exhibit proactiveness and coordinated responsiveness. Proactiveness means that the environment should anticipate problems based on early reports from sensors and take steps to avoid or mitigate them; and coordinated responsiveness means that the overall environment should react as a whole, defending the system as a whole against large-scale, correlated problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures.

### 5.2 VEGA ARCHITECTURE

Vega is composed by a semantic service bus that connects all the elements, a set of components that realize the basic activities present in event-driven architectures, namely event detectors, event handlers (and as we shall see event brokers placed inside the semantic service bus); agent adapters that allow the connection of external software; and the Firenze Environment, a design tool used to create the knowledge level descriptions and annotate services (using the process described in 4.5 Firenze Framework Design and Annotation Process).
5.2.1 Service Bus and Semantic Service Bus

The *semantic service bus* is composed by a “conventional” service bus that is surrounded\textsuperscript{14} by a set of semantic-aware intelligent event brokers that use knowledge as a medium to achieve self-management (concretely self-configurability and self-optimization) and scalability (see Figure 5-3). The nature of this knowledge is diverse and includes knowledge about the services that handle events, the domain where the events occur and the events themselves.

The backbone of the semantic service bus in Vega is a conventional service bus. *Service buses*\textsuperscript{15} are standard-based integration platforms that combine messaging, Web services, data transformation, and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications (Chappell, 2004). They can be considered as the old message and queues systems refurbished with brand new Web service technologies, and more importantly, with all its associated standards (see (Alonso et al., 2003)).

\textsuperscript{14} From a physical deployment perspective, they are placed on top of the conventional service bus.

\textsuperscript{15} The concept of “Service Bus” used in this work is equivalent to the widespread notion of “Enterprise Service Bus”. We prefer the use of the former, as we believe it to be more proper name for a type of infrastructure that is not at all bounded to enterprise-related scenarios.
As is shown in Figure 5-4, the service bus provides the basic communication infrastructure to elements that are situated upper in the architecture. The top-level event brokers (event broker\textsubscript{A}…event broker\textsubscript{E} in the picture) are linked using doted lines, but this association just represents the set of brokers that an event broker knows about; the physical communication is performed uniquely by the service bus.

Let us enumerate the main characteristics and benefits of an architecture deployed as a service bus, which has been observed in its implementations when deployed in industrial scenarios, implementations such as Mule\textsuperscript{16}, Apache ServiceMix\textsuperscript{17}, OpenESB\textsuperscript{18}, PEtALs\textsuperscript{19}, etc. In our opinion these characteristics make them the best choice for deploying an event-driven architecture, and they are:

- **Dynamic and distributed routing.** The service bus can alter its topology altering the routing of messages, since routing information and policies are expressed declaratively and are modifiable. Furthermore the activity of routing is carried out in a distributed fashion; the routing of messages is carried out in a decentralized way. Thus, the message routing performance bottleneck and robustness problems are lessened.

- **Dynamic deployment and configuration driven implementation.** New modules can be plugged directly, without disrupting other operations. This feature, added to the option of changing routing information, brings the possibility of adapting the whole system with little or no operational interruptions; making easier module replication.

---

\textsuperscript{16} http://www.mulesource.org  
\textsuperscript{17} http://servicemix.apache.org  
\textsuperscript{18} https://open-esb.dev.java.net/  
\textsuperscript{19} http://petals.ow2.org
- **Distributed governance framework.** It is possible to deploy and manage the whole lifecycle of the myriad of services connected to the service bus in a centralized manner; even though they are probably remotely and dispersely located.

- **Message queuing based architecture.** Message queuing systems are the cornerstone of loosely coupled service-oriented architectures (Kayne, 2003); as they have been well studied for decades as enablers of the asynchronous communication of elements that shape a distributed system infrastructure. Moreover, message queuing systems have proven great scalability capacity, which is a key feature for our system.

- **Robustness.** As a direct consequence of its distributed and dynamic nature, service-oriented systems deployed as a service bus exhibit great fault tolerance since they can handle module replication and error detection/recovery easily. Monitoring services can be plugged to the bus to check the state and function of vital modules; and if one of these modules goes down, another module of the same characteristics can be directly plugged in the bus just by redirecting all the messages that had to be sent to the crashed module to the new one.

### 5.2.2 Agent Adapters

An agent is any external entity, either human or software, that wishes to interact with the environment or provide additional functionality (e.g. management modules, monitoring and provenance tools, and a long etc.), since the modules described in this dissertation are merely infrastructural. Agent adapters listen to their external messages and applications-internal events and invoke the messaging system in response to these events (they are a realization of the Channel Adapter Pattern presented in (Hohpe and Woolf, 2003)). In other words, they can be used to connect external software agents to Vega. We will not enter in further details since from an internal perspective an event adapter is exactly the same as an event handler (that will be described in detail soon). From an external perspective, all the logic depends completely of the domain and the nature of the agent, whether it is a software agent, a human user, etc. Either case it is domain dependent and falls outside of the definition of the description of this architecture.

### 5.2.3 Design Tools

Vega is a knowledge-based system that relies heavily on the formal description of its components, both domain ontologies and descriptions carried out with the Firenze Framework. Therefore, we consider that design tools should be included as part of the main components of the architecture. For developing ontologies, there is a variety of software available, and most of them generate ontologies implemented following common standards. For developing Firenze
Framework descriptions we have created the Firenze Environment, we will briefly describe its main characteristics in this section.

Firenze Environment tool extends the work in ODESWS (Gómez-Pérez et al., 2004b); later extended in SGSDesigner (Gómez-Pérez and González-Cabero, 2006). Firenze Environment is the front end and graphical user interface that enables user to perform the Virtual Organization formalization design and the later service annotation processes using the Firenze Framework (as depicted in the section 4.5 Firenze Framework Design and Annotation Process). For the sake of clarity we represent a summary of this process in Figure 5-5, where we represent in a compact manner all the process steps. This process begins with the creation of the knowledge level description (step A in Figure 5-5). The Firenze Environment provides a graphical user interface that enables the creation of knowledge level descriptions in a graphical fashion using the graphical notations of each of the elements that belong to the Firenze Framework. Once this graphical knowledge level description of the entities compose the Virtual Organization is created, it must be related with the entity that we wish to annotate (step B in Figure 5-5). Once again, the Firenze Environment provides a graphical user interface for this task; the creation of the mappings is made in a high-level graphical fashion. Finally, once we have related this entity with a knowledge level description, it can be automatically translated into different representational instantiations (step C in Figure 5-5) using a variety of translators that the Firenze Environment provides.

![Figure 5-5 The Firenze Environment process.](image-url)
The main characteristics of this environment that enables users to perform this process are:

- **Knowledge level descriptions are made in a graphical fashion.** The user is not aware of the underneath representation of the models presented in the Firenze Framework. The whole creation of designs, and the annotation of services are carried out just by drawing or dragging and dropping knowledge components. All these graphical definitions are grouped as workspaces. Workspaces are similar concept to projects in software development tools. They contain the definitions of knowledge components and provide all the mechanisms and diagrams for defining them. Workspaces are basically composed by trees and views:

  - **Views** allow users to specify all the features of the knowledge components that describe, and they are represented as tabs. For each top element of the ontologies of the Firenze Framework (e.g. tasks, methods, services, agents, etc.) several views are defined, in order to define graphically its different models in an orthogonal way. The graphical representations to define these models mimic exactly the diagrams that we used to depict them in the Firenze Framework.

  - **Trees** show the hierarchy of the knowledge components that are currently defined in the environment. Trees are very useful in order to define views, since they allow dragging and dropping the knowledge components directly in to the views, where we relate them.

- **Semantic mark-up export capable.** Once the knowledge level description of the services and Virtual Organization has been created a symbol level representation of them should be automatically generated depending on the user desires and needs. Currently OWL-S (OWL-S, 2004) and the RDF(S) OntoGrid service model, but new translators can be plugged-in with ease.

- **Multiple and heterogeneous ontologies handling.** As we have already stated, this environment uses ontologies developed using other ontology engineering tools. We try to be the less restrictive as possible with these ontologies. Currently we support OWL and RDF(S) ontologies stored in files or available in any instance of the WebODE ontology workbench.

- **Able to annotate the main services description language.** The design tool should be able to annotate the most important mainstream technologies and standards for describing services. Currently it is able to annotate WSRF (Czajkowski et al., 2003) and WSDL 1.1 (WSDL 1.1, 2001) services descriptions, but its modular structure allows the seamless inclusion of new languages annotators.
5.2.4 Event-driven Architecture Modules

On the top of the service bus (as depicted in Figure 5-4), we situate three classes of event-driven modules, namely Event Brokers, Event Handlers, and Event Detectors. Each of these types of components realizes what we consider the pillar activities of event-driven architectures, namely event handling, complex-event detection, and event brokering. Let us describe them from the bottom to the top:

- **Event brokering.** This activity consists in the intelligent delivery of event notifications to those modules that are capable of processing them; or those that have expressed their interest on them. As we shall see, in Vega the other basic activities depend heavily on this activity, this is why we place it underneath them in Figure 5-6.

- **Complex-event detection.** This activity involves the detection of certain combination of events observed in a precise time window. On the one hand this activity permits the detection of global states in the system just taken into account small but significant changes. On the other hand, it facilitates the subscription mechanism that prevent the overwhelming of certain modules which are interested in few events, and may be overwhelmed by the huge number of simple event that may occur in the system.

- **Event handling.** This activity represents the response of the overall system towards certain event (simple or composite) received in (or produced inside) the system. Depending on the type of the event, it should be processed by different event handlers, which are the elements that encapsulate the functionality of the system (in the shape of semantic services, as we shall see). As in the case of the complex event detection, this activity relies on the event brokering activity, in order to deliver event notifications to event handlers that are able to handle them.

For each of these activities we define its own subsystem, whose architecture will be described with great detail in the following sections, that is 5.3 The Event Handling Architecture, 5.4 The Event Brokering Architecture, and 5.5 The Event Detection Architecture.
5.3 The Event Handling Architecture

Once an event has been observed in the system, the environment should react carrying out the appropriate actions. As we are building a service-oriented architecture, a proper reaction to the happening of a given event notification means that the proper set of services consumes it. Therefore the event handling activity involves firstly that the notification arrives to the right event handlers (which wrap semantic services, as we shall see); and also includes the treatment of that notification once it arrives to the right set of event handlers, either consuming it, transforming it, or generating a completely new one.

In case of the handling of event notifications in Vega, we extend the concepts that were initially present in SEDA (Welsh, 2002). SEDA (Staged Event-Driven Architecture) is a proposal for event-driven architectures\(^{20}\) that decomposes a complex event-driven application into a set of stages performed by a set of event handlers connected using queues. SEDA tries to move away from the high overhead thread-based concurrency models introduce, decoupling event and thread scheduling from application logic (Welsh, 2002). Finally, SEDA also introduced is a novel approach to avoid service\(^{21}\) degradation, proposing a per-stage request admission control.

The limitations and drawbacks that prevent us from using SEDA as off-the-shelf solution are:

- **Static deployment.** The main limitation of SEDA is its static deployment dependent configuration. SEDA usage scenarios are static, in the sense that the functionality of the system is solely based in the ordering of the stages and the links between the handlers. This configuration is static and made at design time. We propose, as we have already described in 5.2.1 Service Bus and Semantic Service Bus, the use of a semantic service bus that allows the dynamic communication between event handlers.

- **Lack of explicit semantics.** As we described in the knowledge-based system principle (see section 5.1.1 Distributed Knowledge System Paradigm) in order to raise the level of intelligence of the modules of the architecture, especially in the case of event brokers, we need formal models shared among the architecture components. Thought SEDA handles some statistical information extracted from handlers response times, as we shall see, it

\(^{20}\) SEDA is not strictly an event-based architecture in the sense that we describe in this work. It is event-driven in the sense that its activities respond to external asynchronous events, but it does not follow the rest of the principles that we identify for event-driven systems.

\(^{21}\) Service in the broader sense of the word.
Vega handles only low-level information about notifications and the domain where the environment is about to be deployed. To the contrary, we propose the definition of formal models about the events that might be handled and the domain where the event-driven application is to be deployed.

In Figure 5-7 we depict how the original SEDA event handlers were deployed and how they are deployed in Vega. Originally in SEDA event handlers topology was created at design time and could not be altered afterwards (we represent the topology as arrows that relate event handlers in Figure 5-7). It is important to realize that this immutable ordering of the event handling stages greatly determines the functionality of the system. In Vega we just plug all the event handlers in semantic service bus, as we represent in Figure 5-7. Instead of presetting the event handlers topology (fixing thus the order of the execution) the delivery of events is done on the basis of the formal description of the events and the service that provides the functionality of each of the event handlers. The right event handler receives the notifications that it is able to handle thanks to the semantic service bus, and more importantly, this matching is made at runtime (how it is achieved is described in great detail in section 5.4 The Event Brokering Architecture).

In summary, Vega event handlers are settled around the semantic service bus. In the case that the event handler transforms an event, or in the case that it creates a completely new one, the event handler simply delivers the new event notification to the bus, which becomes responsible
of its delivery to any other component in the architecture interested in that notification. With this approach to event handling we achieve the following benefits:

- **Loose coupling.** The semantic bus adds a level of indirection that makes the functionality of each event handlers completely isolated of the rest of the event handlers in the environment. But although they do not know about each other, the bus takes care of organizing a collaborative response.

- **Points of failure removal.** We lessen functional coupling among event handlers and henceforth we avoid potential points of failure. The middleware is the responsible of delivering the notifications to services, in case that one of these services goes down, the overall system remains functional.

- **Runtime dynamism.** The system is able to perform late binding deciding upon different criteria (e.g. functional properties, non-functional properties, workload, etc.) in a seamless fashion.

### 5.3.1 The Event Handler in Detail

In this section we take a closer look at inner details of a single event handler. Event handlers are wrappers that allow the connexion of a semantic service instance to the semantic service bus. As depicted in Figure 5-8 we consider the event handler as divided in two separated facets. Its **external perspective**, composed by the observable behaviour (described by the internal service instance interaction process, see 4.3.2.2 Interaction Process) and two URIs that address the input and output mailboxes of the event handler; and the **inner part** of the event handler, composed by the mailboxes, its knowledge base, its controller, its internal state and last but not least the core of its functionality, its service instance. In the following paragraphs we describe each of these elements, which we portrait in Figure 5-8.
External elements. External elements are those that provide information essential for external entities to interact with the event handler. They are:

- **Observable behaviour.** As we discussed in the section devoted to services description in the Firenze Framework chapter (section 4.3 Services Model), we consider services as reactive systems. As such, the interaction with the service is stateful and possesses several and complex flows of control. Future interactions depend on the observable state of the service, and therefore its interaction context must be explicit and accessible. Moreover, this description of the interaction should be formally described, so that external agents can interpret it. As the core service instance of the handler is a service described with the Firenze Framework, the interaction process of the service instance describes the observable behaviour of the event handler. It depicts the flow of control, its execution context and the way it handles exceptional situations.

- **Input and output mailboxes URIs.** In order to receive and deliver event notifications, we must assign a unique identifier for both event handler input and output mailboxes.

Internal elements. These elements are not externally accessible to other entities of the environment. We make such a strict distinction since if any other element of the system were aware of its content, and it made its decision on its basis, it would result in undesirable functional coupling. And coupling is one of the most common causes of scalability problems. We consider the following internal elements for an event handler:
• **Input and output mailboxes.** The inputs and outputs of an event handler (i.e. notifications) are received and stored in its input and output mailboxes respectively. That means that event notifications are not directly delivered to the instance of the service, but are delivered to the input mailbox of the handler. When appropriate, the notification is delivered to the service. Each of the mailboxes posses a monitor that registers when a notification has entered the queue, and labels the notification with that timestamp. This information will be used by the controller to estimate the state of the event handler as we shall see.

• **Knowledge base.** Each of the event handlers, in appliance of the lack of complete knowledge and distributed decision making principles (one of the architectural principles described in section 5.1 Vega Architectural Principles), posses and uses its own knowledge base. The knowledge base acts as a memory that stores the set of facts that the event handler needs to know to carry out its problem-solving actions. It can be modified and updated by the operations that the service carry out when consuming the incoming notifications. Therefore, the knowledge base holds a mix of both the initial model knowledge (likely unmodified); and also non-monotonic knowledge about the execution context that the event handler gathers. The service operations decisions are solely based upon the notification event that it is processing and facts stored in the knowledge base.

• **Service instance.** In the core of an event handler resides an instance of a semantic service. Event handlers are components, which means that they are design to be used without change in different domains. Components provide infrastructure, not domain specific functionality. This kind functionality is provided by a hosted service instance in their core that provides specific functionality the application domain. This service instance is formalized using the Firenze Framework.

• **Controller and internal state.** The function of the controller is to monitor the performance of its associated event handler in order to determine whether is functioning normally, or detect a possible overload or underuse situation (what we refer as internal state). The approach that we follow is a generalization of the one described in (Welsh and Culler, 2003)(Welsh, 2002), further extended to support the detection of other kind of situations and the appliance of different policies when an undesired situation is detected.

The controller observes the response time for each of the notifications, which is the time that takes a notification to go from the input mailbox to the output mailbox.
Therefore we consider the time that the service needs to process the notification and the time that the notification stays in the queue. About the first time we cannot do much about it, the duration of the services operation is something that we cannot control (but we must take it into account). The time that the notification stays in the input queue ready to be consumed might be more significant, as it is a clear indication that the service is overloaded. And moreover, sending fewer notifications to the event handler, which is something that is in our hands, can lessen this situation. Nevertheless, the solely action that the controller performs is to update the internal state of the event handler; any other actions to alter the environment in order to avoid undesired situations should be done by other module that observes this state.

The controller adapts the method presented in (Welsh and Culler, 2003)(Welsh, 2002) for determining the state of the event handler. This method consists on a set of activities performed in two different phases.

- **In the initialization phase.** The controller waits a number of requests in an initialization state waiting to have enough significant time samples. The number of request is determined by the constant `initializationSetSize`.

- **In the working phase.** The controller keeps on recollecting samples of the time it takes for each notification to travel from the input to the output mailboxes. The size of the sample set is either determined by a size parameter, `numberOfSamples`, or a time out parameter, `timeToSample`, whatever comes first. These samples go thorough a smoothing filter to prevent that unusual spikes might make the controller overreact to exceptional non-representative observations. The expression of that filter is the simplest form of an exponentially moving average with
parameter $\alpha$ ($0 < \alpha < 1$). The $i$-th element is obtained from a combination of the previous filtered sample and the actual sampled value (where the initial value $current_0$ corresponds to the first sample)

$$current_i = sample_0$$

$$current_i = \alpha \text{sample}_i + (1-\alpha) current_{i-1}$$

Then, as described in (Welsh and Culler, 2003), the controller then determines the 90th-percentile value of the sample set, and compares it with a target value of the expected time that should take to accomplish the operation. To do so, the controller orders the samples from the lowest to the highest, and it picks the one that corresponds to the sample set size multiplied by 0.9. Then it determines the discrepancy of that time (we refer to it as $current_{90th}$) with the targeted one (i.e. $target$ in the following formula)

$$discrepancy = \frac{current_{90th} - target}{target}$$

Depending on whether $discrepancy$ is higher than the upper allowed discrepancy (represented as $error_{up}$, where by definition $error_{up} > 0$); or it is below the lower allowed discrepancy ($error_{down}$ which by definition $error_{down} < 0$) the internal state of the event handler must be updated accordingly, as shown in Figure 5-9. The target time parameter (and the admissible errors) should be ideally determined by the provider of the service instance hosted in the event handler as part of its non-functional properties. Nevertheless, it can be easily derived if we include a training phase for the system. Using a well-studied notification set we can determine by statistical methods the value of the normal time interval for each event handlers (i.e. $[target(1+error_{down}), target(1+error_{up})]$).

---

22 There is no formal procedure for choosing $\alpha$. Statistical technique may be used to optimize the value of $\alpha$, we will initially use 0.7, that was determined as optimal in (Welsh and Culler, 2003)

23 According to (Welsh, 2002) the 90th-percentile was chosen as a realistic and intuitive single-valued measure of client-perceived system performance. Other options, such as the average or the maximum response time fail according Welsh to represent the shape of a response time curve.
5.4 THE EVENT BROKERING ARCHITECTURE

Once an event notification arrives or is produced inside an event-driven system, the system must react in a suitable manner. So far, we know how the notification is created/consumed/transformed in the system by event handlers. But complementary to this activity, we must provide the necessary mechanisms to deliver these notifications to these event handlers. More generally speaking, we must provide mechanisms to deliver automatically the notification to the set of event handlers that have the ability, and the capacity in that precise moment, to handle them. This last clause is of great importance, since it asserts that the proposed architecture strictly follows the design principles that we wished for an event-driven system (see these principles in section 5.1.3 Event-driven Paradigm Principles and part of those proposed in 5.1.4 Autonomic Computing Principles). Applying the afore referred principles we expect to cope with the challenges that we pose for the brokering subsystem, which are the following:

- **Expertise based routing.** We extend and generalize the concepts proposed in the SWAP architecture (Haase et al., 2008). Routers form a semantic overlay network, independent of the network topology (i.e. the backbone service bus in our case). Events are forwarded to the peers (i.e. other brokers) that a given broker knows that may able to handle them; we refer as those which expertise is similar to the content of the event. The expertise of a peer is the semantic description of the knowledge base of a peer, and will be described in detail in the rest of this chapter.

- **Achieve autonomic behaviour.** As we have already stated in the principles section, one of the objectives of our architecture is to achieve self-configuration. Currently, the configuration activity involves a lot of human interaction. Routing information, in current content-based routers, consists of huge routing tables, XPATH expressions, rules, etc. If the environment is simple, the time and complexity of updating the routers might be affordable. But in the case of large systems, or small but very dynamic ones, a quick system administrator or XML guru would not be enough to configure it. The layer of event brokers automate this routing configuration, its endeavour is to become almost transparent to the upper layers of the architecture and users.

- **Lessen coupling.** In most of event-driven or message oriented applications a publish-and-subscribe model of interaction is employed. Components subscribe to receive messages about certain topics. If the hierarchy of topics changes, components should be modified to reflect these changes. This is an undesirable case of coupling between components and the system. In our approach, the component is deployed and the system is responsible of finding
out which are the events that should be delivered to the component. If there is any change in the topics hierarchy, the client is completely unaware of them.

- **Lower events traffic.** The timeliness requirement of event-driven systems implies a huge quantity of events. We must prevent flooding the bus with additional traffic that event routing may cause (i.e. unnecessary hops on notifications delivery should be minimized).

- **Implicit workload distribution.** Depending of different non-functional properties, message routing may be updated to distribute in an optimal way the workload of the different modules. Though it is not one of our primary objectives, we avoid sending events to modules that may not be able to handle them diligently.

- **Decentralize of routing knowledge base.** Decentralized routing approaches are proven to increase scalability and robustness (Curry, 2006). To achieve so, an efficient and effective distribution of the routing knowledge base between the involved routers is a must.

The routing system is composed by a set of brokers. A *broker* is a component responsible of delivering event instances to components that are plugged to the bus. This activity should be carried out:

- **Efficiently.** Notifications should be delivered in an efficient manner, using knowledge about the components and the events to deliver (these topics will be described with great detail later).

- **Transparency.** Brokers should not perform any change on the notifications they deliver nor they should generate new ones. Moreover, they should use just the content of the notification plus the description of the event handlers; they should be completely transparent to events originators/receivers.

In the following sections we answer what we consider the big questions about the routing of event notifications by means of peers expertise.

- **Which are the elements involved in the routing of events?** We should not only enumerate them, but we should also provide a clear definition of each element. This topic will be covered in 5.4.1 Model for Expertise-Based Event Routing.

- **How do we represent and create expertise?** Each event handler wishing to receive/create event notifications in the system delivers the precise and formal description of the events it is able to consume/produce to a local broker. How do we represent this information? How we will create it? Does it change depending on the type of brokers?. We describe all this topics in the section 5.4.2 Expertise Knowledge Representation and Creation.
How do we route an event notification? Each of the routers delivers notifications to the set of brokers in its neighbourhood that the router knows that is able to handle them properly. As handling a notification properly, we mean that it can either consume or deliver it directly or indirectly to a final event handler that can consume it\textsuperscript{24}. The question is how the broker determines which from its neighbour peers expertise description are able to handle it. We depict how a broker will route an event notification using the expertise of its neighbourhood in the section 5.4.3 The Expertise-Based Routing Algorithm.

5.4.1 Model for Expertise-Based Event Routing

Let us define the formal model for expertise-based notification routing. As we have stated, our starting point are the elements and abstract SWAP architecture (Haase et al., 2004)(Haase et al., 2008) that proposes using expertise knowledge for distributing semantic queries in a network of knowledge bases in an intelligent manner.

![Expertise based routing model.](image)

In the following subsections, we describe all the entities that we consider necessary to describe the event notification routing problem, namely peers, shared model (composed by the events model and the domain model), expertise, notifications, subject, and the abstraction and matching processes. These entities are portrayed in Figure 5-10, plus the relationship that exist among them. For each of them we outline the differences and extensions that we propose to the

\textsuperscript{24} This is in line with the efficient broadcast communications principle that we have stated in the section 5.1.3 Event-driven Paradigm Principles; we should avoid event notification flooding as much as possible, since it is by definition a bad policy for resource management.
original entity described in the SWAP architecture (which for the shake of clarity we describe in the section 8.4 Appendix III: SWAP, A Distributed Peer-to-peer Knowledge Base).

5.4.1.1 Peers

Event brokers are organized as a peer-to-peer network. Contrary to the SWAP architecture, we propose an asymmetric peer-to-peer model, as we consider different peer types that handle different types of information about their neighbour brokers (or event handlers). In this subsection we focus on introducing these brokers that as shown in Figure 5-11 can be of three different types, namely local brokers, border brokers and inner brokers. We have taken this acyclic topology and taxonomy of peers from REBECA (Fiege and Mühl, 2000). Nevertheless, as the routing mechanisms, the representation of notifications, and the representation of peers that we propose are completely different from those described in (Fiege and Mühl, 2000)(Mühl et al., 2006) we just borrow from REBECA the peers nomenclature and their situation inside of the topology. Let us describe them:

- **Local brokers.** They are used to connect the event handlers to the semantic bus. Thus their main endeavours are on the one hand recollecting the events notification of these event handlers. On the other hand, they bring together its formal descriptions delivering it in a compact form. As shown in Figure 5-11, they communicate only with border brokers.

- **Border brokers.** They are responsible of routing notifications that flow from/towards local brokers. They are also responsible of grouping the descriptions that local brokers deliver (we describe this activity with great detail in the section 5.4.2 Expertise Knowledge Representation and Creation).

- **Inner brokers.** They are connected to either other inner broker or to border brokers having no direct connection with external entities. They are in charge of delivering notification on
the basis of the expertise of its contiguous brokers and spreading intelligently the expertise descriptions that border brokers provide. The basic difference with other brokers is that they might perform relaxed delivery of notifications, as we depict in the section 5.4.3 The Expertise-Based Routing Algorithm.

![Diagram of event broker main components.](image)

Figure 5-12 Event broker main components.

In Figure 5-12 we show the structure of an event notification broker. As the reader has surely noticed, their structure is basically identical to the structure of an event handler. Notifications to be routed are delivered in the input mailbox, like in the case of event handlers. That allows an asynchronous interaction with this component; the broker can consume the notification on its own convenience. Once it has been, the broker puts the notification in its output mailbox, so that the communication layer delivers it. Event brokers, like event handlers, also posses a controller that monitors continuously the event broker to determine it state (i.e. initializing, normal, overloaded, and underused). Let us the remark their differences:

- **Notifications are not modified.** The content of each event notifications stored in the output mailbox remains unaltered. Instead this type of component, using its knowledge base and applying the expertise-based matching process, tags these notifications with the set of brokers URIs that should receive this notification. The underneath infrastructure, the physical communication layer (i.e. the service bus), takes this notification out of the mailbox, and then deliver it to the input mailboxes of the set of recipients it has been tagged with.

- **The component functionality is fixed.** Event handlers are components that provide different functionality depending on the domain specific service instance that is hosted within them (see Figure 5-8). To the contrary, event brokers behaviour is always basically the same, to carry out the routing algorithm that we present later on (section 5.4.3 The Expertise-Based Routing Algorithm)
- **The content of the knowledge base.** The body of knowledge of a broker is solely composed by the expertise of the surrounding peers that the broker is aware of (i.e. its neighbourhood expertise).

### 5.4.1.2 Shared Model

In the definition of expertise we mention that it is expressed in terms of a shared model. The simple domain-dependent ontology shared among peers considered in the SWAP architecture might be expressive enough routing of queries about the conceptualization described by that shared ontology. After all, the routing system just has to deliver one type of notification, which is a request for query answering. However for delivering several types of event notifications we must consider a richer semantic model that takes into account several notification types, and different types of relationships among them. We need to consider thus two models:

- **Events model.** As we have described in Chapter 4 Firenze Framework, for each Virtual Organization we identify a set of event ontologies (all of which belong to the events model presented in section 4.4.3 Events Submodel). Their concepts are the different types of events (both simple and composite) that may occur inside the environment eventflow. The expertise will be in part expressed in terms of these ontologies.

- **Domain model.** In the formalization process proposed in the description framework the environment is described in terms of a set of domain models (see section 4.2.3 Domains,). As such, they should also be considered in the definition of the elements that belong to the expertise based routing model.

### 5.4.1.3 Expertise

The delivery of events across the bus is realized in terms of peers expertise and the subject of the notifications. The concept of expertise was initially defined in (Haase et al., 2004) as an abstract, semantic description of the local node repository of a peer. This description is based and expressed on terms of a shared ontology. SWAP architecture focused in the intelligent forwarding of queries on the basis of semantic similarity. The rationale behind SWAP architecture is that in order to achieve broadcasting queries to all peers should be avoided; the forwarding of events should not be made to all or a random set of known peers, but only to the ones that have a good chance of answering it (Haase et al., 2004). This definition of expertise, as it is focused solely in the routing of semantic queries, makes the following assumptions:

- **Peers only route queries.** Brokers (which are the knowledge repositories) just route one type of notification (i.e. queries); and also there is only one type of peer node (i.e. knowledge repositories).
Peers only generate answers to queries. There is always a direct correspondence relationship between the messages that a node may generate and the ones that it has received in the past.

For an event-driven system scenario these assumptions are neither feasible nor realistic. Thus we extend the expertise definition; we define expertise as an abstract, formal description of the interests and future actions of a peer inside an event-based system. As action we understand the delivery, creation or consumption of a notification of an event. This description is expressed in terms of the shared model (i.e. domain model plus event model). With this definition of expertise we extend SWAP architecture notion of expertise as the knowledge that a peer possesses to the description of the actions that the peer can do in the environment. The use of such richer shared model introduces the possibility of defining several types of notifications that reflect different situations of interest. Finally, we want introduce the concept of expertise covering. We say that the expertise of a peer A covers the expertise of a peer B, if each action contained in the expertise B implies that this action is also contained in the expertise of the peer A. The concept of expertise covering turns to be very important in order to lessen redundancy, as we shall see in the expertise creation algorithms.

As we want to carry on describing all the entities that describe the event notification routing problem in a compact manner, let us postpone the thorough description of expertise. Later on in this chapter we provide a thorough description of expertise and the use that we make of it, including the description of the different types of expertise according to its purpose and nature, how we generate it, and which representation structures and reasoning mechanisms we use to represent and make use of it for routing notifications.

### 5.4.1.4 Notification

A notification is an instance of an event, it is a concrete record of an activity in a system (Luckham, 2002), the detection state change in a computer system (Mühl et al., 2006) (for a precise description of how we events describe events we remit the reader to section 4.4.3 Events Submodel).

### 5.4.1.5 Subject

A subject is an abstraction of a given notification (i.e. event instance). This abstraction is created in terms of the shared model that we have afore described, and is created to be able to compare the notification with the expertise of a peer (and therefore determine whether the subject notification is covered by the expertise of a peer or not).
5.4.1.6 Abstraction Process

The abstraction process consists on the one hand on the creation of the expertise descriptions from peers; and on the other hand, in the extraction of the subject of the notification. As shown in Figure 5-10 this process lifts the abstraction level in order to achieve a better and more meaningful comparison between notifications and peers ability to handle them.

5.4.1.7 Matching Process

Brokers carry out the matching process once they receive a notification. This process consists on determining which brokers of its neighbourhood (i.e. the peers it knows about) should receive this newly received notification. This process is realized invoking a matching function that given the expertise and the subject obtained by the matching process, evaluates whether the notification should be delivered to the peer or not.

5.4.2 Expertise Knowledge Representation and Creation

As we have already defined, we consider expertise a formal description of the interests and future actions of a peer inside an event-based system. In this section we describe a classification of the different types of expertise that we define in our architecture (section 5.4.2.1 Types of Expertise); the structures that we will allow us to represent expertise at the symbolic level, which will allow using these different types of expertise reasoning with them (section 5.4.2.2 Binary Decision Diagrams for Expertise Representation); and finally, how we are going to create this symbolic representation from knowledge level representations (section 5.4.2.3 Expertise Descriptions Representation).

5.4.2.1 Types of Expertise

In this section we classify expertise according to two criteria, its purpose and the peer type that it describes. According to its purpose we distinguish two types of expertise, publication and subscription expertise. According to the peer type that describes, we distinguish between event handler, local broker, border broker and inner broker expertise. We shall define all of them in the following subsections.

5.4.2.1.1 According to Purpose

The basis of the expertise concept as we have stated was set in the SWAP architecture. SWAP architecture focused on the efficient forwarding of queries. Peers, after evaluating the query, could only generate a result to the query that was sent back to the original query originator. In Vega brokers forward different types of notifications that may be produced by
different kinds of peers. Moreover those peers might have previously shown their interest in receiving just certain type of events; and as one of our principles is the efficient broadcast of notifications, we should only deliver them notifications of that type. We should thus extend the notion of expertise so that it distinguishes between publication expertise and subscription expertise. More in detail:

- **Subscription expertise** is knowledge about the events that a peer is interested in receiving to act upon them. This expertise is the basis for routing notifications, since notifications should be delivered only to components that can handle them. Once a peer receives a notification of an event, depending on the peer nature it may consume it (in the case of event handlers) or delivering the event to other component (in the case of any other type of broker).

- **Publication expertise** is knowledge about the future actions of a peer in the event-based system. As action we consider the delivery or creation of a notification in the system. In the case of event handlers this expertise specifies the events that in the future an event handler might generate inside the eventflow. In the case of the different types of broker, this facet of expertise describes the events that the component may forward to its adjacent neighbourhood.

### 5.4.2.1.2 According to the Peer Type

As we have previously presented in section 5.4.1.1 Peers, we identify different types of peers that are of different nature. Since expertise of the peers is a formal description of the actions of such peers, we differentiate between different types of expertise. Although peers perform the same action (deliver event notifications using knowledge about their surrounding peers), brokers handle knowledge of different nature to describe such actions. Let us enumerate each of the peer types and describe the type of expertise (we summarize this enumeration in Figure 5-13):

- **Local broker expertise.** The local broker expertise is the union of the descriptions of the event handlers plugged to the bus, that is, the sum the formal descriptions of the functional capability of the service instance that a local broker hosts. We refer to this description of the capability also as the event handler expertise; despite of not being peers, the description of the capability of an even handler is an analogous to the concept of the expertise of a broker.

- **Border broker expertise.** In a border broker the concept of expertise is twofold:
  
  - **The internal view of the border broker expertise** can be considered as the union of the expertise of each of its subscriber local brokers.
  
  - **The external view of the border broker expertise** is exposed toward the set of inner brokers, and can be considered a sieved perspective of this expertise, lets call it blurred expertise. This expertise is more abstract and also imprecise, but it reduces the
complexity of handling it. This topic that will be thoroughly described in the section 5.4.3 The Expertise-Based Routing Algorithm.

![Diagram](https://via.placeholder.com/150)

Figure 5-13 Different types of expertise of different types of brokers.

- **Inner brokers expertise.** The inner brokers expertise is somehow abstract indirect expertise, since they handle the external expertise provided by border brokers. And as we have said, this expertise might have been dimmed to some extent. This expertise can be obtained both in a direct manner (i.e. announced by a directly connected border broker) or indirect way (i.e. announced by other inner broker).

### 5.4.2.2 Binary Decision Diagrams for Expertise Representation

Hitherto we have just sketched abstractly what expertise is, we have been agnostic regarding its representation. Let us now in this section specify how we are going to represent expertise; and which are the mechanisms that are going to be used to make effective use of it. It is recommended for the reader who is not familiar with binary decision diagrams to read section 8.2 Appendix II: Binary Decision Diagrams devoted to them.

In (Haase et al., 2008) the peers share a unique ontology $O$, which is used for describing the expertise of peers (that is also used for extracting the subject of queries that are sent through the network of peers). In this approach, the expertise ($E$) of a peer ($P$) is defined as $E \subseteq 2^C$ where $C$ denotes a set of concepts that belong to the afore-mentioned shared ontology. For example, in the case of (Haase et al., 2005) the shared ontology is the ACM topics taxonomy; the expertise of a peer is in consequence a set that belongs to the power set of the ACM topics taxonomy. A topic belongs to the expertise of a peer if and only if that peers has instances of that topic.
This method of representing expertise has a number of limitations for the description of expertise about actions and not about the instances that a knowledge base might have. The problem with the expertise representation proposed in (Haase et al., 2004)(Haase et al., 2008) is simply that authors aimed at different target than us, since their aim was to represent knowledge bases populated by individuals. These individuals could belong to a discrete set of concepts and in that concrete case the expertise of a peer can be considered the set of concepts that a given peer holds. A service instance cannot be categorized following this approach solely, since their description does not only contain information about what they are or what they are not, but about their capability. Henceforth the representation should be more expressive and capable of representing entities in a more fine grained way; we cannot just use vector of Booleans that relates the individual with the set of classes it might belong or not. As the expertise of one peer can be viewed as a summary of the descriptions of all the services that it is aware of, it consequently suffers from the same limitations as the description of one single service.

Therefore, we need a richer representation to describe peers expertise. We have chosen to use binary decision diagrams, which are graph-like data structures used to represent a Boolean functions. In the section 8.2 Appendix II: Binary Decision Diagrams we include the basic theory about binary decision diagrams and their use in routing scenarios in order that the reader could understand what binary decision diagrams are and how we plan to use and extend them for our purposes. Nevertheless, let us enumerate the variety of reasons that made us choose them, among the more:

- **They have been very well studied for decades.** Both the structures to represent them and their associated algorithms. Initially they were a revolution in CAD and VLSI applications (because they are a canonical and efficient way to represent and manipulate Boolean functions). Since this initial use, binary decision diagrams as have been successfully used in numerous fields such as model checking, knowledge bases, symbolic machine learning, and a long etcetera. Finally, as was firstly proposed in (Campailla et al., 2001), they have also been successfully applied in conventional event routing scenarios.

- **Their structure.** Their structure makes them interesting in different ways:
  - **Compact representation format.** It is a type of representation that allows the definition of complex expressions in a compact manner, if they just obey some simple rules (see section 8.2 Appendix II: Binary Decision Diagrams).

---

25 Canonical means that for a given variable order there is one and only one ordered binary decision diagram for each of the possible Boolean functions.
• **Tree-like structure.** They are structured as a tree-like, structure that makes them layered if we avoid repetition in their nodes. As we shall present this property allows representing incomplete Boolean expressions in a systematic way, just taking into account the expression that the binary decision diagram represents just to a certain depth.

In the following sections we describe in detail the different types of binary decision diagrams that we use. In the case of event handlers we use state of the art binary decision diagrams, as we describe in section 5.4.2.2.1 Binary Decision Diagrams for Representing Event Handlers. For representing other types of expertise we propose two different extensions to binary decision diagrams, topics that are covered the section 5.4.2.2.2 High-level Binary Decision Diagram for Representing Expertise.

**5.4.2.2.1 Binary Decision Diagrams for Representing Event Handlers Expertise**

The formal description of an event handler is the description of its hosted service instance. These services instance is described using the Firenze Framework, which is very useful for the design phase. As we stated in the section 3.2.1 Knowledge level Description Framework, describing services at knowledge level is very useful to make meaningful statements about a system behaviour without reference to the structures and reasoning mechanisms. But once we have finished the design phase and we want to deploy and use these descriptions, we must lower this description to a concrete symbol level representation, and choose the suitable reasoning mechanisms in order to use it. As we represent in Figure 5-14, from the abstract service instances descriptions of the Firenze Framework we generate specific symbol level structures, binary decision diagrams; and we define concrete reasoning mechanisms to use them, the matching function (we describe this function later on in the section 5.4.3.2 The Matching Process). In the case of event handlers we propose to use state of the art binary decision diagrams (we describe them in detail in section 8.2 Appendix II: Binary Decision Diagrams).
5.4.2.2 High-level Binary Decision Diagram for Representing Expertise

In this section we will depict an extension to binary decision diagrams in order to make them suitable to represent the different types of expertise that we have depicted so far. We will define high-order binary decision diagram as the tuple of sets \( \langle N, A \rangle \), which represents a rooted, directed acyclic graph. More in detail:

- **The set \( N \) of nodes.** The tree contains three types of nodes:

  - **Atomic formula nodes.** We consider each node as an atomic subformula, formalized as the triplet \( \text{Node} = \langle \text{Variable}, \text{Operator}, \text{Value} \rangle \) (with its respective functions such as we define for each node in a binary decision diagram). Each node will have associated several functions, such as \( \text{variable} : \text{Node} \rightarrow \text{Variable} \), \( \text{operator} : \text{Node} \rightarrow \text{Operator} \), \( \text{value} : \text{Node} \rightarrow \text{Value} \), and the new function).

  - **Formula nodes.** These nodes represent arbitrary complex formulae, represented by binary decision diagram (either first or high-order binary decision diagrams).

  - **Terminal nodes.** As in the case of any other binary decision diagram, we define two terminal nodes labelled \( 0 \) and \( 1 \). Obviously, \( 0 \) corresponds to false and \( 1 \) corresponds to true.

- **The set \( A \) of arcs.** The set of arcs \( \{ \langle \text{Node}, \text{Node} \rangle \} \) is determined by two functions \( \text{low}(\text{node}) \) (i.e. \( \text{low} : \text{Node} \rightarrow \text{Node} \)) and \( \text{high}(\text{node}) \) (i.e. \( \text{high} : \text{Node} \rightarrow \text{Node} \)). These functions return in a given node which will be the next node depending on whether the atomic formula holds (the high function) or not (the low function).
5.4.2.3 Expertise Descriptions Representation

So far we have given a precise definition of what expertise is, the different types of expertise that we distinguish, and what the different types of representation mechanism that we are going to use. In this section we describe the representation of expertise in terms of these representation mechanisms. First we describe the representation of the expertise of local brokers (5.4.2.3.1 Local Brokers Expertise Representation), and then the expertise of both border and inner brokers (5.4.2.3.2 Border and Inner Brokers Expertise Representation).

5.4.2.3.1 Local Brokers Expertise Representation

We can think of the local broker expertise as the aggregation of the subscription/publication expertise of all the event handlers that are connected to the broker. Let us firstly describe how we represent the expertise of event handlers, so that we can later show how we represent the expertise of a local broker as their aggregation.

As we depicted in the section devoted to event handlers, their core domain-dependent functionality is provided by a service instance. This service instance formally describes using the Firenze Framework (the section regarding service description can be found in section 4.3 Services Model). Recall Figure 5-8, where we show the service at the core of the event handler and its operations. These operations that consume, modify, or create events notification are functionally described using problem-solving methods. In order to route notifications to event handlers, brokers should rely only on invariant and externally observable functional descriptions. Methods describe internal details that should not be taken into account by external agents when discovering services in terms of their capability, since it would introduce an unnecessary coupling. The external state of the service instance should not either be taken into account. As it varies in time, it can change whilst the notification travels thorough the semantic bus several times, and we cannot assume that each broker is aware of the actual state of the event handler on each of the routing steps. The inclusion of state knowledge in the notification delivery decision process would violate the lack of complete knowledge principle (see section 5.1.1 Distributed Knowledge System Paradigm), which will for sure result in a scalability killer on the one hand; and on the other, due to its dynamic nature, this knowledge may be even result misleading.
Therefore, as is depicted in Figure 5-15, for representing the event handler expertise we use binary decision diagrams that formalize the competence of the task associated with the service. In consequence, these binary decision diagrams are a formal representation of the functionality of the service.

The local broker expertise is the aggregation of the subscription/publication expertise of all the event handlers that are connected to the broker. From a knowledge-oriented perspective, we can say that the expertise is the set of event handlers the local broker knows about. From a logical point of view it can be represented as

\[ \text{LocalBrokerExpertise}_i = \text{EHExpertiseExpression}_0^i \lor \ldots \lor \text{EHExpertiseExpression}_N^i \]

Each of the \( \text{EHExpertiseExpression}_j^i \) correspond to the logical expression of the expertise of the event handlers. As we consider two types of expertise, according to its purpose, this expression must be divided in the following expressions:

\[ \text{LocalBrokerPublicationExpertise}_i = \text{EHPublicationExpertiseExpression}_0^i \lor \ldots \lor \text{EHPublicationExpertiseExpression}_N^i \]
and

\[ \text{LocalBrokerSubscriptionExpertise}_i = \text{EHSSubscriptionExpertiseExpression}_i^0 \lor \ldots \lor \text{EHSSubscriptionExpertiseExpression}_i^N \]

where each \( \text{EHSSubscriptionExpertiseExpression}_i^j \) and \( \text{EHPublicationExpertiseExpression}_i^j \) corresponds to the logical expression of the expertise of a concrete event handler.

Figure 5-16 shows graphically these topics, the representation in terms of binary decision diagrams. Initially we have the expertise of each of the event handlers separated in publication and subscription expertise. The resulting expertise is the aggregation of all the individual event handlers expertise description, where each event handler expertise is expressed in conjunctive normal form. Therefore, we create a high-order binary decision diagram that embodies the union of the expertise description of some of the event handlers that the broker hosts. Nevertheless, not all the event handlers are used to represent the expertise of the local broker, and perhaps we might need to make some transformation on them. Each of the nodes labelled as \( \text{EHS}_i \) or \( \text{EHP}_i \) in Figure 5-16 are binary decision diagrams that represent an event handler expertise, using equivalent expressions to their expertise description but in conjunctive normal form. Moreover, we will only include nothing more than the event handlers that are strictly necessary. Let us depict these topics more in detail:

- **There is no redundant event handler representation.** All of the event handlers nodes \( \text{EHS}_i \) and \( \text{EHP}_i \) in Figure 5-16 are indispensable in order to represent the expertise of the broker. That means that if we take out any of them, the set of notifications depicted by the local
broker expertise is altered. As we shall describe in section 5.4.2.4.2 Brokers Expertise Representation Creation, the construction of such representation is incrementally performed, in order to make it in an efficient way. We check if the expertise description of an event handler is already covered before inserting a new node (i.e. if the set of notifications that the node represents is not already contained in the set represented by the broker expertise representation).

- **Event handlers expertise representation are in conjunctive normal form.** In section 5.4.2.4.1 Event Handler Expertise Representation Creation we described how we create the formal description of the expertise of an event handler (both for publication and subscription, recall Figure 5-15). In the creation of the local broker expertise we need to create logical expressions equivalent to those, but expressed in conjunctive normal form (in case that they are not already expressed in such way, of course). A logic formula is expressed in conjunctive normal form if it is expressed as a conjunction of clauses, being each of these clauses a disjunction of literals. In short, a logical formula is expressed in conjunctive normal when it is expressed as follows:

$$
\bigwedge_{i=0}^{N} \bigvee_{j=0}^{M} \text{Formula}_{ij}
$$

In this concrete case, we express the expertise of each of the event handlers inside the logical expression of the expertise of the whole local broker as follows:

$$
\text{EHExpertiseExpression} = \bigwedge_{i=0}^{N} \bigvee_{j=0}^{M} \text{EventHandlerFormula}_{ij}
$$

where each $\text{EventHandlerFormula}_{ij}$ is an atomic formula (or its negation), that we represent as a triplet $\langle \text{Variable, Operator, Value} \rangle$ in a binary decision diagram.

Expressing each of the event handler expertise in conjunctive normal form assures that each of these descriptions share the same structure, which is precisely what makes it so interesting for our purpose. We take advantage of this structure for relaxing these expressions in a sound way, as we will describe in the section devoted to the routing algorithm (5.4.3 The Expertise-Based Routing Algorithm).
In the picture we show the structure of the expertise of an event handler publication expertise (of the node $EHP^M$ that corresponds to the $EventHandler^M$ publication expertise). The conjunction of disjunction is expressed using a high-level binary decision diagram, being each of its disjunctions the nodes labelled as $F^M_0$...$F^M_N$ in Figure 5-17. This binary decision diagram represents thus the following expression:

\[ EHP^M = \bigwedge_{i=0}^{N} F^M_i \]

Each $F^M_i$ has associated its own binary decision diagram, since they contain disjunction operators and therefore cannot be treated as atomic formulas. Nevertheless, they are pretty simple, and can be expressed as the disjunction of atomic formulas, that is:

\[ F^M_i = \bigvee_{i=0}^{N} AF^i_j \]
5.4.2.3.2 Border and Inner Brokers Expertise Representation

Structurally, inner brokers and the blurred border brokers expertise is the same as the local broker expertise.

As shown in Figure 5-18, the expertise description from border/inner brokers are extracted from the expertise of its neighbourhood; a set of local brokers, in the case of border brokers, and of a mixed set of border and inner brokers, in the case of inner brokers. The characteristics of the expertise are exactly the same of those presented in the last section (i.e. 5.4.2.3.1 Local Brokers Expertise Representation). Therefore, each of the expertise representation nodes $EHS^i$ and $EHP^j$ are expressed in conjunctive normal form; and none of these event handler expertise representation nodes is redundant.

5.4.2.4 Expertise Representation Creation

In the following sections we depict how to create the different types of expertise representations that we described in the last sections. For the shake of clarity we separate in two sections this process, we start by depicting how the expertise representation of the event handler is created separately, taking as a starting point Firenze Framework service descriptions (section 5.4.2.4.1 Event Handler Expertise Representation Creation); and after that we describe how we create the brokers expertise representation incrementally taking into account event handlers descriptions (section 5.4.2.4.2 Brokers Expertise Representation Creation).
5.4.2.4.1 Event Handler Expertise Representation Creation

The generation of the expertise binary decision diagram is made using the well-known algorithm presented in section 8.2 Appendix II: Binary Decision Diagrams. This algorithm generates a reduced binary decision diagram from a logical expression. We do not impose any restriction in the ordering of the variables when creating individual handlers description due to the size of descriptions. The aforementioned algorithm using a random variable ordering, results in efficiently manageable binary decision diagrams, as we handle rather small ones. If you translate the representation of an individual service functional description to a binary decision diagram, the resulting structure posses such small number of nodes that finding the best order would not result in a significant benefit. We remit the reader to classics of symbolic model checking that used binary decision diagrams, such us (Burch et al., 1998), where they were already handling efficiently binary decision diagrams of a number of nodes that ranged from $10^5$ to $10^6$ nodes; whereas a service instance operation capability description can result in a binary decision diagrams that might range around 10 to 100 nodes.

As shown in Figure 5-15, we generate two binary decision diagrams, one for subscription expertise and another for publication expertise. Both of them are extracted from the competence of the task that describes the service instance of the event handler, but taking as starting point different parts of it.

- **Subscription expertise.** The subscription expertise binary decision diagram is obtained from input roles type expressions and the precondition of the task, since they describe the events that the event handler will consume.

- **Publication expertise.** The publication expertise is extracted from the output roles type expressions and its postcondition, since they describe the events that the event handler generates.

5.4.2.4.2 Brokers Expertise Representation Creation

The key concepts for the creation of brokers expertise representation is that it is created in an incremental fashion as new event handlers are added (or removed) from the semantic service bus. The basic operations that each broker must implement are thus are the addition/removal/update of the description event handlers. We follow such incremental approach as we assume a very low churning rate on brokers (and a fixed topology); and we can assume that there might be a high churning rate for event handlers. Nevertheless the operation of the removal/update/addition of a broker is still possible; it translates in applying a
removal/update/addition operation for each event handler expertise expression in the broker expertise.

![Figure 5-19 Brokers operations for the incremental creation of expertise representation.](image)

Aligned with the incremental approach that we have presented, in Figure 5-19 we represent the messages that brokers exchange to build and keep consistent their expertise representation. For each type of message, brokers implement the homonym operations, so in this section we might use both terms indistinctively. Basically for each operation we have to specify how the knowledge base of the broker is altered (i.e. the internal formalism that the broker uses to perform the brokering activity); and how the expertise of the broker (i.e. the external formal description of the broker) is affected. More precisely:

- **Knowledge base update.** Event brokers create a representation of its expertise and this is specifically what other brokers know about them. Nevertheless, the broker stores additional information in its knowledge base that is used when creating expertise.

- **Expertise cover graph.** Brokers store a graph which nodes are the event handlers expertise descriptions, and its arcs the cover relationship. There might be event handler expertise representation nodes that are not part of the expertise description of the broker, since they might make it redundant. Despite this fact, we cannot get rid of them since in case of the removal of the node of the expertise description that covers its expertise, we might need to place them back in the expertise description of the broker.

- **Service types.** Local brokers associate for each of the description of an event handler the type of the service instance that they hold. This information prevents the broker from creating the expertise of that service class more than once, and is useful to sieve redundant expertise descriptions.
o **Expertise update.** In some cases, the operations for creating expertise described update this expertise. Therefore, we must specify how it would be changed, and the messages that need to be sent to its neighbours so that they can update their knowledge about the broker.

Let us specify the expertise related operations for each type of broker in the following sections, section 5.4.2.4.2.1 Local Broker Expertise Representation Creation for the local brokers and section 5.4.2.4.2.2 Border and Inner Brokers Expertise Creation for both border and inner brokers.

5.4.2.4.2.1 Local Broker Expertise Representation Creation

The local broker, as depicted in Figure 5-19, carries out two different operations to create its expertise description. One, add(EH) in Figure 5-19, is used when a new event handler is registered in the system; and other, remove(EH) in Figure 5-19, deals with the contrary situation, when an event handler is removed from the system. Let us describe each of them:

- **Event handler addition.** This operation is performed when a new event handler is registered in the environment. We decompose this operation in the following steps:

  - **Event handler expertise representation creation or retrieval.** When adding a new event handler the first step is the creation or the retrieval of its expertise representation. If the local broker has any other event handler registered that holds the same type of service, it is not necessary to create the representation again, it is simply retrieved. If not, the local broker should perform the process described in the section 5.4.2.4.1 Event Handler Expertise Representation Creation.

  - **Event handler description addition.** Once the expertise description of the event handler has been created, we add it to the local broker expertise. Before adding a new handler description we have to check whether this expertise description is not already covered by any other event handler expertise description present in the broker expertise description. For determining whether an event handler expertise covers other event handler expertise, we propose to use the cover function described in (Li et al., 2005). This covering test leads to three possible situations:

    - **Other event handlers EH_i...EH_j that belongs to the broker expertise cover the expertise of EH_new.** In such case, the new node with the event handler expertise representation is not added, as it would lead to a redundant expertise representation. The following actions are taken:
Update the knowledge base of the broker. The expertise cover graph must be updated, $EH_{new}$ must be added to the adjacency list of $EH_i$ as it is covered by it.

The expertise $EH_{new}$ is not added to the expertise representation of the broker.

- The expertise of $EH_{new}$ covers the expertise of other event handlers expertise $EH_i...EH_j$ that belong to the broker expertise. The new event handler expertise covers others that are already part of the broker expertise representation. Therefore, as we seek a non-redundant expertise representation, we must act as following:

  - The knowledge base of the broker is also updated. In the expertise cover graph, the covered event handlers $EH_i...EH_j$ should now be part now of the adjacency list of the new event handler $EH_{new}$.
  
  - The new event handler expertise representation replaces the ones that it covers in the expertise representation of the broker.
  
  - The broker sends an update message to its surrounding brokers so that it updates the description changing the nodes $EH_i...EH_j$ with the more general expertise representation of the new event handler $EH_{new}$.

- Otherwise. If none of the above statements applies, the event handler expertise representation is simply added to the expertise representation of the broker. That translates in to:

  - The new event handler expertise representation is added to the expertise representation of the broker.

  - A message $add(EH_{new})$ message is propagated to the neighbourhood of the broker.

- Event Handler Removal. The removal of an event handler from the expertise of a broker may lead to three different possibilities:

  - The event handler description $EH_{toRemove}$ does not belong to the broker expertise description. That means that the expertise of $EH_{toRemove}$ is covered by the expertise of an event handler $EH_i$ that does belong to the expertise description of the broker. Therefore:

    - The knowledge base of the broker is updated. $EH_{toRemove}$ expertise representation is removed from the expertise cover graph.

    - No further actions are required. As the event handler description was not part of the expertise representation of the broker, none of the neighbour brokers knew about it.
• **The event handler description** \( EH_{\text{toRemove}} \) **belongs to the broker expertise description and its expertise covers the expertise of the event handlers** \( EH_i \ldots EH_j \). In such case, other brokers are aware of that particular event handler expertise description on the one hand, and there are other event handler expertise descriptions that might need to be added to it. More precisely:

- The knowledge base of the broker is updated. \( EH_{\text{toRemove}} \) expertise description is removed from the cover graph.

- The expertise of the broker must be updated. The update consist on the following actions:
  - We must firstly determine the nodes that could have been replaced by the node \( EH_{\text{toRemove}} \) when it was originally added to the broker expertise. These nodes should be the expertise descriptions that were covered by the expertise of \( EH_{\text{toRemove}} \) and were not covered by any other event handler expertise. This set is easily determined by means by the expertise cover graph. Every node that was part of the adjacency list of \( EH_{\text{toRemove}} \) that are not in the adjacency list of other event handler expertise are our candidates. Let us denote this set of candidates \( EH_i \ldots EH_j \) (Note that this set could be empty).
  - The expertise representation of the broker is updated. The node that represented expertise of \( EH_{\text{toRemove}} \) is replaced with those of \( EH_i \ldots EH_j \).
  - The update messages must be sent to all the surrounding brokers to notify the change. A message \( \text{update}(\{EH_{\text{toRemove}}\},\{EH_i \ldots EH_j\}, \text{Broker}) \) is sent to the neighbourhood of the broker.

• **The event handler description** \( EH_{\text{toRemove}} \) **belongs to the broker expertise description and its expertise does not cover the expertise of any event handlers.** The event handler expertise representation is directly added to the expertise representation of the broker. That translates in the following actions:

- The new event handler expertise representation is added to the expertise representation of the broker.

- A \( \text{remove}(EH_{\text{toRemove}}) \) message is propagated to the neighbourhood of the broker.
5.4.2.4.2.2 Border and Inner Brokers Expertise Creation

In the case of the border and inner brokers, they both create the expertise representation using the same operations. The basic operations are, as we have already discussed, the addition/removal/update of event handlers all the event handlers of the local broker. The new operation that we have to describe is the update operation, but it is a mere combination of addition and removals.

- **Adding and removing event handlers expertise representation.** The procedure for the addition/removal of a single event handler is exactly the same as in the case of the local broker. The only difference is that local brokers have to create the representation of the expertise, whereas border and inner brokers receive this representation as parameter.

- **Updating expertise.** This operation is just a concatenation of add/removal operations, but we pack them together in order to assure that these operations are carried out in an atomic manner. If we send the add and remove separately, other modifications might occur in the middle of the update, what might leave the expertise of the broker and its knowledge base in an inconsistent state. Therefore, when we perform an update operation, let's note it $update(\{EH_1...EH_i\},\{EH_k...EH_l\})$, the broker performs an add operation for each expertise description in $\{EH_1...EH_i\}$; and a remove operation for each expertise description in $\{EH_k...EH_l\}$. And whilst these operations are performed, any other message remains unattended in the input mailbox of the broker until the update is finished.

5.4.3 The Expertise-Based Routing Algorithm

So far we have described how the description of the expertise is created. Let us present now the routing algorithm that, making use of this symbolic representation, routes the notifications in an intelligent manner. We propose a innovative approach to event notification routing which rationale is basically to combine in a gradual fashion the flooding of events in cases where a broker is completely overwhelmed; with precise delivery in cases where routers are in an optimal state.
Figure 5-20 Rationale behind the proposed routing algorithm.

- **Thorough the innermost brokers zone.** As shown in Figure 5-20, all the notifications travel thorough the innermost brokers. It is not hard to observe that these brokers are likely to sustain a greater load. This problem is even worse since the average size of the descriptions of peers that we propose is considerable larger and complex than state of the art subscription-per-subscriber approaches, what prevent us from using state of the art merge-based algorithms (see (Mühl et al., 2006) for a survey of such techniques). The rationale behind merge-based algorithms is that for a set of subscription entries we can create a merger that can substitute several subscriptions, making routing tables smaller and simpler to evaluate. There are two broad types of mergers, namely imperfect mergers and perfect mergers. In the case of perfect merger, the set of the notifications that match the merger is exactly the same as the notifications recognized by the set of its constituent filters. Imperfect mergers, to the contrary, guarantee that at least they covering of all the notifications that were matched by the original set of filters, but they also might match other notifications that were not originally matched (i.e. the notification recognized by the original set of filters are a subset of those recognized by the merger). In our concrete case, our peer descriptions are complex, what makes the computational cost of creating and deleting the merges automatically cumbersome (specially for the perfect merges case). Moreover, this operation is likely to be performed quite frequently since we assume a possible high event handler churning rate (i.e. the formed merges should be continually recalculated. Our target is to achieve scalability, and with the length of the expertise-base subscriptions, innermost brokers will be drawn evaluating expertise descriptions that might not even lead to accurate notification delivery. So we propose an algorithm that trades certain grade of uncertainty to obtain a decrease in evaluation complexity; uncertainty that should be possible to graduate instantaneously with no computational effort, even for
complex descriptions. This grade of uncertainty would make the matching process less complex, allowing innermost brokers to handle individual notifications faster, lessening their probability of ending overloaded.

- **As we get closer to the edges of the routing system.** Towards the edge of the routing system the number of notifications is reduced drastically, which means that brokers can take more time to improve the precision of their matching process. Finally, no matter the overload that they might have, the outer brokers (local brokers in our architecture) must carry out a precise matching of the notifications, since event handlers should only receive notifications that they can cope with.

After stating the rationale behind our routing algorithm, let us describe it. We decompose it in two processes, the abstraction process and the matching process. These processes were defined in a very high-level fashion in the model for expertise-based routing in Figure 5-10 (see section 5.4.1 Model for Expertise-Based Event Routing). Let us define them thoroughly for the concrete case of our routing algorithm.

### 5.4.3.1 The Abstraction Process

The abstraction process consists both in the creation of the expertise descriptions of the peers and the extraction of the subject of the notification. As we have already described thoroughly the creation of expertise (see 5.4.2.4 Expertise Representation Creation), in this section we focus on the creation of a notification subject.

![Abstraction Process Diagram](image)

Figure 5-21 The abstraction process.

We consider a notification an instance of an event. As such, it is an instance of at least one event defined in the events model (and also its description might reference to domain model as shown in Figure 5-21); and consists on a set of attribute assignations (in the case of instance
attributes), and a set of assignment that might be implicit (such as the case of the class attributes of the event concept it is instance of). The subject of a notification, as we use binary decision diagrams for representing expertise, becomes a binary decision diagram variable assignment extracted from the notification. In this assignment we consider both its explicit attribute values, and the implicit that can be inferred. Thus, as pictured in Figure 5-21, in the creation of the variable assignment that embodies the subject of the notification we do not only consider the content of the notification, but the semantics of the overall event and domain models.

Extending the approach described in (Campailla et al., 2001), we create a variable assignment that can be later used to evaluate the binary decision diagrams that represent the expertise of peers. We formalize the subject of a notification as the following variable assignment:

\[
\text{subject}(\text{var}) = \begin{cases} 
\text{val} & (\text{var}, \text{val}) \in S \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

where

\[
S = \{(\text{val}, \text{var}) \mid \text{Domain Model, Event Model, notification} \models (\text{var}, \text{val})\}
\]

5.4.3.2 The Matching Process

Each broker carries out the matching process when a notification is delivered to its input mailbox. This process basically consists on checking which brokers of its neighbourhood (i.e. the set peers it knows about) should receive this newly received notification. The broker therefore needs to invoke a matching function that given a peer’s expertise and the subject of a notification, evaluates whether the notification should be delivered to the peer or not.

As we consider the subject a variable assignment, and we describe expertise using binary decision diagrams, the matching function becomes the evaluation of the binary decision diagram
with the variable assignation defined by the subject. In other words, each matching function represented in Figure 5-22 can be written as:

\[ \text{matching function}_i = \text{evalBDD}(\text{subscription expertise}_i, \alpha_{\text{notification}}) \]

This use of binary decision diagrams for notification routing is based on the one proposed in (Campailla et al., 2001) that has been later on used by different systems (see section 8.2.1 Binary decision diagrams in publish/subscribe systems). Nevertheless we have extended it, since there are certain differences that initially might appear subtle, but make our problem completely different from the individual entity publish/subscribe approach taken by other event-driven systems. We handle brokers expertise, which translates in the description of a node with a great expressivity and at the same time great granularity. This makes a huge difference of the aforementioned approaches, which focus on describing publishers/subscribers individually and using less expressive languages.

![Figure 5-23 The three axis of complexity of the matching process.](image)

Therefore we have carried out a detailed study of the sources of complexity that emerge in our concrete case of matching process, and we propose different approaches to tackle them. We depict this graphically in Figure 5-23 where the sources of complexity of the matching process algorithm are associated with the solution we propose. We consider the complexity of the matching algorithm as a three dimensional problem, where each of the axis corresponds to one of the following subsections. That includes the excessive number of neighbours on the
knowledge base of the broker (subsection 5.4.3.2.1 Excessive number of neighbours); the excessive length of each formula that represents a peer expertise (subsection 5.4.3.2.2 Length of the expertise formulae in the entries); and the complexity of evaluating each of the nodes of the expression (subsection 5.4.3.2.3 The complexity associated with evaluating each of the nodes).

5.4.3.2.1 Excessive number of neighbours

One of the basic problems of event-driven systems is that initially any external agent that might generate or consume an event notification should announce its interests and capabilities to the system. Therefore, in a naïve approach these descriptions are flooded in to the system, resulting in each of these entities requiring its own entry in the routing table. This approach on the one hand makes the routing table almost impossible to handle due to its complexity (both time and space complexity); but also violates one of our architectural guiding principles, the lack of complete knowledge principle (described in section 5.1.1 Distributed Knowledge System Paradigm). Many techniques have been proposed to overcome this problem; in our concrete case, we have taken a broker expertise-oriented approach, which assumes that we are not going to send directly the subscriptions or advertisements of single event handlers, but the expertise of brokers. In consequence, the number of average entries in the system is reduced drastically as it becomes the average size of each broker neighbourhood.

5.4.3.2.2 Length of the expertise formulae in the entries

In the previous subsection we stated that the broker expertise approach reduced the problem of having an excessive number of peers in a broker neighbourhood. The drawback of this approach is that the average length of the expressions that describe expertise is longer. In order to avoid an excessive length we combine two techniques, one that takes into account the cover relationships among subscriptions (the non-redundancy property that we imposed in the brokers expertise, see 5.4.2.3 Expertise Descriptions Representation); and other that relaxes binary decision diagrams to make them shorter.

The length of the subscription is directly related and proportional to the expressivity of the language used to represent subscriptions/publications. This situation is worsened as we propose an expertise-based routing model. Instead of managing event handlers individually, we group their descriptions in using expertise of brokers. Even though we propose the use of non-redundant structures (see 5.4.2.3.2 Border and Inner Brokers Expertise Representation) the

---

26 Senso stricto, our expertise-based approach in terms of “classical” event-driven systems can be considered partially as part of the perfect merge algorithms family.
length of the expression that represent such expertise will be even greater. Obviously, the longer
the expression is, the more precise the subscription we; but as seems obvious, it takes time to
evaluate it.

Figure 5-24 The concept of blurred expertise and the matching function.

To overcome this problem we define the concept of blurred expertise, which is used by inner
brokers. **Blurred expertise** is represented in the same manner that we have described so far for
inner and border brokers expertise. The difference resides in that when using peer’s expertise to
evaluate whether a given notification should be delivered to the peer, the inner broker do not
always consider the whole expertise expression, instead they just take into account part of it.
And is precisely the part that we consider what we refer as blurred expertise. As shown in
Figure 5-24, the level of detail variation is applied per event handler expertise description, and
is determined by the depth parameter, becoming the logical expression of an event handler
(subscription in this case) expertise:

\[
EHS^N_{(depth)} = \bigwedge_{i=0}^{depth} F_i^M
\]
Where the parameter \( depth \) ranges from 0 to the actual size of the binary decision diagram, and each \( F_0^N \ldots F_M^N \) represent a disjunction of atomic formulas (see 5.4.2.3.2 Border and Inner Brokers Expertise Representation).

As we reduce the depth parameter, the expression of the expertise is relaxed. That results in a reduction of the complexity of performing the matching function, but also causes a relaxation on the set of notifications that the expertise expression represents. More precisely:

- **Complexity reduction.** Obviously, if we shorten the conjunction that represents each of the event handlers \( (EHP_N) \), it takes less time to evaluate them. Each of them represents a conjunction of one to several atomic formulas; the removal of them makes the event handler, and the overall broker expertise, less complex to evaluate.

- **Notification set relaxation.** We relax the notification in the sense that, if we note the set of notification that make valid the expression \( EHP_{(depthA)}^M \) up to a depth \( depthA \) as \( N(EHP_{(depthA)}^M) \), we can state that

\[
\text{depth}_A > \text{depth}_B \rightarrow N(EHP_{(depthA)}^M) \subseteq N(EHP_{(depthB)}^M)
\]

We can assert such a statement because we have expressed the expertise each event handler in conjunctive normal form. The conjunction of disjunction is expressed using a high-level binary decision diagram, where each of the disjunction is represented as a one of nodes labelled as \( F_0^N \ldots F_M^N \). As we relax the formula decreasing the depth parameter, we reduce the number of clauses in that conjunction. And since the clauses are ordered, each time we remove a clause we know for sure that the resulting expression will yield a more generic expression than the previous one. Moreover, if we consider the expression of the expertise of a broker, not only a single broker, as we define it as the disjunction of the expertise of the event handlers

\[
\text{BrokerExpertise}_{\text{depth}} = \bigvee_{i=0}^{N} EHP_i^M_{\text{depth}}
\]

we can generalize the relaxation of event handlers expertise to brokers expertise, and we can write

\[
\text{depth}_A > \text{depth}_B \rightarrow N(\text{Broker Expertise}_{N(depthA)}) \subseteq N(\text{Broker Expertise}_{N(depthB)})
\]

As a conclusion we can say that as we reduce the depth of an expertise descriptions when matching notifications, the set of positive notifications perhaps contains some false positive in terms of the strict evaluation of expertise. But the important points are on the
one that we will not lost any notification that belongs to the notification set of the original expertise description. And on the other hand, false positives are gradually filtered and removed as they approach brokers that are closer to the boundaries since these brokers use less relaxed expertise representations, what prevent local brokers from being overwhelmed by them (in the extreme case of one false positive reaching such brokers, it not be delivered to the subscribers, as local brokers do not relax the descriptions)

Finally, we must specify how the depth of the matching algorithm is calculated, which is part of the self-configuring facet of the presented architecture. The depth parameter of the algorithm is determined in terms of the internal state of the inner broker (see 5.3.1 The Event Handler in Detail for a complete description of how the internal state of a module is determined in Vega). After an initialization phase, we assign the length of the longest path depth. In other words, initially there is not any relaxation at all. Then, after an initialization time, each time that the controller evaluates the internal state of the broker, it tunes (if necessary) the level of relaxation of each inner broker using the following function:

\[
update(\text{relaxationLevel}) = \begin{cases}
\text{relaxationLevel} + 1 & \text{currentState} = \text{Overloaded} \\
\text{relaxationLevel} - 1 & \text{currentState} = \text{UnderUsed} \\
\text{relaxationLevel} & \text{otherwise}
\end{cases}
\]

The \text{currentState} of each event broker is determined individually by the respective controller of each event broker using the algorithm presented in section 5.3.1 The Event Handler in Detail. The \text{relaxationLevel} represents proportion of each node expertise description that the broker prunes in the matching process. The depth that is considered on each expertise node (i.e. \text{EH}_i) in the matching process is the following:

\[
\text{consideredDepth} = \text{depth}(\text{EH}_i) \times (1 - \left\lceil \frac{1}{\text{maximumRelaxationLevel}} \right\rceil \times \text{relaxationLevel})
\]

where \text{maximumRelaxationLevel} is a constant fixed before the brokering system starts functioning, and represents the different levels of that a inner broker. We initially propose to work with five different levels that range from no relaxation to ignoring the expertise description (i.e. \text{maximumRelaxationLevel} equals 4) but this is a parameter that can be later specifically tuned on each Vega deployment depending on the granularity that we wish to
achieve in the different relaxation levels, and the average size of the event handlers expertise descriptions.

The evaluation function for binary decision diagrams (we include the original in 8.2 Appendix II: Binary Decision Diagrams) that takes into account the depth parameter is defined as follows:

$$\text{evalRelaxedBDD}(\text{bdd}, \alpha_{\text{assignment}}) = \text{evalNodeRelaxed}(\text{root}(\text{bdd}), \alpha_{\text{assignment}}, \text{consideredDepth})$$

$$\text{evalNodeRelaxed}(\text{node}, \alpha_{\text{assignment}}, 0) = \alpha_{\text{assignment}}(\text{node})$$

$$\text{evalNodeRelaxed}(1, \alpha_{\text{assignment}}, _) = \text{true}$$

$$\text{evalNodeRelaxed}(0, \alpha_{\text{assignment}}, _) = \text{false}$$

$$\text{evalNodeRelaxed}(\text{node}, \alpha_{\text{assignment}}, \text{depth}) =$$

$$\left[ \alpha_{\text{assignment}}(\text{node}) \land \text{evalNodeRelaxed}(\text{high}(\text{node}), \alpha_{\text{assignment}}, \text{depth} - 1) \right]$$

$$\lor$$

$$\left[ \neg \alpha_{\text{assignment}}(\text{node}) \land \text{evalNodeRelaxed}(\text{low}(\text{node}), \alpha_{\text{assignment}}, \text{depth} - 1) \right]$$

Basically this evaluation function is conceptually equal to the one of conventional binary decision diagrams, but when the evaluation reaches a given depth it does not evaluate further, and takes as leaf nodes the last evaluated nodes (see Figure 5-24 for a graphical representation).

### 5.4.3.2.3 The complexity associated with evaluating each of the nodes

The matching algorithm must evaluate each node in the binary decision diagrams that we use for matching once a notification arrives. To address that we use the concept of global predicate-index depicted in (Li et al., 2005) to avoid the repetition of realization of unnecessary computations. In consequence, though the high-level binary decision diagrams appear to be a huge data structures that are to be evaluated sequentially, each atomic expressions contained in this diagrams is only evaluated once and stored in a cache data structure. Therefore the evaluation of this tree can be seen at low level as the partial evaluation of the flattened array that contains all the atomic expressions in the nodes of the binary decision diagram; plus the evaluation of the and/or relationships between these elements that is different for each complex formula.
5.5 THE EVENT DETECTION ARCHITECTURE

Event-based environments are by nature highly distributed and decoupled. As stated in (Vera et al., 1999) in such systems the notion of global state is difficult to grasp. This is why complex-event detection is so important in an event-driven architecture, since composite events are effective for representing environment states in a very compact manner (as presented in (Romer and Mattern, 2004)). The reason is that they do not picture the whole state of the system, but they focus on representing significant state changes in relevant intervals of time.

When designing the complex-event detection architecture, apart from following the generic architectural principles that we enumerated in section 5.1 Vega Architectural Principles, we intend to achieve the following more specific goals:

- **High-level eventflow oriented architecture.** We must stress this point, since it really affects the whole detection infrastructure. The detection infrastructure addresses the discovery of complex events from the flow of high-level events generated by both agents and services the play a role in the Virtual Organization (as we described in section 4.4 Virtual Organization Model). We expect a discrete flow of high-level events, not a continuous and data-intensive stream of low-level events. For such low-level complex event detection we remit the reader to the work carried out in research initiatives, among the most well known STREAM (Arasu et al., 2003), and Aurora and Borealis (Abadi et al., 2005). These initiatives address the problem of the management of real-time complex event processing and query processing in the presence of continuous data streams. Our approach can be seen as placed on top of these systems; in fact, if we virtualize nodes of these systems using Grid services, they can perfectly be integrated in Vega, becoming these data streams resources handled by a service.

- **Individual global state representation in a specified window of time.** We address at picturing the global state of the system in a given time window. Nevertheless, since we will deploy a set of distributed modules, and we apply the lack of complete knowledge and distributed decision-making principles we expose to external agents the perspective from the point of view from each of the event detectors separately.

- **Low ontology commitment.** Last but not least, we should minimize the ontology commitment made in our approach, both in the representation of complex events and in the mechanisms used to carry out their detection. We should minimize the gap that exist between a knowledge level representation such the one proposed by the Firenze Framework and the actual representation that we will use in the event detection activity.
In order to implement the complex event detection architecture of Vega we follow a similar approach to event handling, since as in the case of this activity its modules (event detectors, $ED_i$ in Figure 5-25) rely heavily on the underneath semantically-enhanced brokering infrastructure (described in section 5.4 The Event Brokering Architecture).

![Figure 5-25 The event detection architecture.](image)

Each of these event detectors subscribes to the simple events that form the complex event descriptions of the complex event they aim to discover, which are stored in the internal part of their knowledge base. The semantic service bus is in charge of delivering all the notifications of these types produced or received in the environment to them, in the same manner they did for event handlers. As the event detectors receive these notifications, by means of the complex event descriptions and the detection mechanisms they search for complex events they know about. In case that one of these events is detected, on the one hand a notification of that complex event is created and delivered to the semantic service bus; and on the other hand the event detector knowledge base external zone is updated accordingly.

In the following sections we depict in detail each of these event detectors. Firstly, we depict its structure, detailing its components functions and relationships (section 5.5.1 The Event Detector in Detail). Then we describe the symbolic representation and the reasoning mechanisms that these event detectors create and use to detect complex events (5.5.2 Eventflow Knowledge Representation and Creation).

### 5.5.1 The Event Detector in Detail

This section is devoted to the description of an event detector. As the reader might realize, its structure is almost the same of the others modules of Vega, the event handlers and event brokers.
In Figure 5-26 we portrait the elements that compose an event handler. Basically they are:

- **External elements.** They are the components of the event detector that expose information essential for external entities to interact with it. In this case they are just the input and output mailboxes URIs. In order to receive and deliver event notifications, we must have assigned a unique identifier to both event handler input an output mailboxes, and this URI must be accessible from the outside.

- **Internal elements.** These elements are not externally accessible to agents outside the environment.
  
  - **Knowledge base.** Each of the event detector (as we depicted in the case of event handlers), in appliance of the lack of complete knowledge and distributed decision making principles, posses its own knowledge base. In the case of the event detector, the knowledge base can be divided it two main corpus of knowledge. One that stores the perspective of the global state of the system on the basis of the complex events that it has detected. This part of the knowledge base can be modified and updated, as we shall see, as the event detector discovers new complex events; and external agents are allowed to consult it. Finally, the knowledge base also includes a rather static corpus of knowledge that contains the descriptions of the composite events that this event detector should detect in the eventflow.

  - **Input and output mailboxes.** The inputs and outputs of an event detector (i.e. the notifications in the eventflow) are received and stored in its input and output mailboxes respectively. Each of the mailboxes posses a monitor that registers when a notification has entered the queue, and stores this timestamp.
- The **output mailbox** stores the notifications that have been detected by the event detector. They stay there until the underlying communication system delivers them; after which they are removed from the mailbox.

- The **input mailbox** can be seen as a FIFO data structure that stores notifications in the same order that they are received. The size of this input mailbox is restricted not by its number of elements, but by the interval of time that its notifications cover. This interval of time is what we refer as the *time window*.

![Figure 5-27 Detail of the event detector input mailbox.](image)

As shown in Figure 5-27, once a notification timestamp falls outside of the interval defined by this time window, it is deleted and therefore discarded.

### 5.5.2 Eventflow Knowledge Representation and Creation

In this section we describe how we represent each of the eventflow patterns that we enumerated in the Firenze Framework, and the mechanisms that we use to detect them. We think that the best way to present this topic is to depict one by one each of. Once again the knowledge level Firenze Framework descriptions are useful at design time; at runtime we need to choose representation and reasoning mechanisms to make use of these descriptions. As shown in Figure 5-28 in order to represent each of the eventflow patterns we propose the use of single-agent abstract state machines; as the main detection mechanism we propose the execution of these abstract state machines.
Abstract state machines (Gurevich, 1988) (Gurevich, 1995) can be thought of as a finite state machine whose states can be viewed as first-order logic structures. Transitions between these states are defined by updates to these structures, specified by means of a transition rules set. In section 8.3 Appendix III: Abstract State Machines we include a description of abstract state machines, so that readers unfamiliar with abstract machines can easily understand the rest of the section. Nevertheless, let us describe the characteristics that made us choose abstract state machines over other possibilities:

- **Flexibility and low ontology commitment.** As we already stated in this section we need a very flexible representation mechanism, since we should not constrain the designs made at knowledge level. The abstract state machines methodology proposes the use of abstract state machine with a very similar purpose; they initially were conceived to allow the refinement from the highest possible level models, composed barely by system requirements, to almost an executable form. This fact makes them a perfect fit for our case.

- **Frame problem handling.** This is one of the key points for the election of abstract state machines. This type of formalism focuses in the updates sets, which translates in the specification of a new and future state by defining just the differences between the present one. The rest of the global state is assumed to remain unaltered. As the reader might have realized, this approach is completely aligned with the event-driven systems paradigm for sketching global states views.

- **Interpretability.** Finally, one of the abstract state machine descriptions is that they can be interpreted and executed (see for example initiatives such as AsmGofer27, ASM Workbench (Del Castillo, 1999), etc.). That means that the descriptions that we are going to propose in the next section provide the detailed steps and an effective manner for detecting complex events.

---

27 [http://www.tydo.de/asmgofer.html](http://www.tydo.de/asmgofer.html)
5.5.2.1 Complex Events Detection

In this section we describe the steps that the event detector should carry in order to detect each of the patterns defined in the Event Model section of the Firenze Framework (see 4.4.3 Events Submodel). For each pattern we define:

- **The detection abstract state machine.** We detail how the detection abstract state machine for detecting the eventflow pattern is created. Therefore we must detail both:
  - **The creation of abstract state machine signature.** Basically we enumerate the functions that the detection abstract machine should posses in its signature.
  - **The creation of abstract state machine transition rules.** In order to represent graphically the rules of the abstract state machine we use automata, in a similar way as described in (Mühl et al., 2006). We use simple finite state machines that have been extended to support temporal relationships, event reception specification, etc. We like specially this approach because of its intuitiveness; anyone with computer science background is familiar with an automaton-like representation.

- **Related activities.** We will also detail any other activity that the event detector must carry out, such as subscription to certain type of event, update of certain functions, etc.

5.5.2.1.1 Property Related Patterns

The basic building blocks for eventflow patterns deal with the reception of simple event and the establishment of relations among the attributes event properties.

5.5.2.1.1.1 Property Discriminator Pattern

The property discriminator pattern (defined in the section 4.4.3.1.1.1 Property Discriminator Pattern) is basically used for receiving notifications of a given event type. Optionally, additional constraints can be defined in order to achieve a finer sieve of the received notifications. In order to perform the activity described in such pattern, we consider necessary:

- **Creation of the abstract state machine.** The creation of the abstract state machine to detect notifications of a given event type involves:
  - **Definition of the functions of the signature.** We should define an in function to store the event notification. Initially it is set to the default value, i.e. `undef`.
  - **Creation of the transition rules.** The set of rules of the abstract state machine should formalize the behaviour presented in Figure 5-29.
Abstract state machine for the detection of an event.

As depicted with the automaton, we simply poll the event function until a notification of these characteristics is received, which means that the function location stores a value different to `undef`.

- **The subscription to the type of event described detector.** In the case of this simple pattern, this is the most important action. The event detector subscribes to the event, adding to the description of the event any additional specific restriction.

- **The update of the event in function.** Finally, the event detector should update the function that stores the event once received (i.e. the in function `event` in Figure 5-28). When inspecting the input mailbox the event detector should inspect the notifications that happen along the interval that goes from the current time to the current time plus the time window.

In situations in which the event detector might have several candidate notifications in its input mailbox we cannot choose among any of the possible candidates, since we would bias completely the detection process, resulting in undetected complex events. Therefore, we create a brand new detection abstract state machine for each possible candidate in the time window; replicating the signature and all the elements of the abstract state machine as they were at the beginning of the inspecting process.
5.5.2.1.1.2 Property Relater Eventflow Pattern

The property relater eventflow pattern (described in detail in section 4.4.3.1.1.2 Property Relater Eventflow Pattern) is used to relate the attributes of different event notifications by means of a set of Boolean relationships. As this is a very generic situation that appears combined with other patterns, likely structural patterns, we check first if detect other more complex patterns. Afterward, we check whether the desired relationships among notification attributes holds. In order to do that, we believe necessary to perform the following steps:

- **Creation of the abstract state machine.** The creation of the abstract state machine to check the relationship among the attributes of certain event notifications involves the following actions:
  
  - **Creation of the functions of the signature.** We should create for each relater defined in the pattern the set of relationship relation functions (\(\text{relationship}_i\) in Figure 5-31).
  
  - **Creation of the transition rules.** The set of rules of the abstract state machine should formalize the behaviour presented in Figure 5-31.

![Abstract state machine for the property relater eventflow pattern.](image)

Figure 5-31 Abstract state machine for the property relater eventflow pattern.

Basically, this automaton resumes what we have already expressed. First of all we detect any complementary situations in the eventflow, and after that we check whether each of the relationships holds (represented in Figure 5-31).

- **The event detector should perform all the activities associated with the event detection automaton.** We are providing a recursive definition; the detection automaton depicted as doted rounded square can be arbitrarily complex. Therefore the event detector must also carry out all the activities associated with it.
5.5.2.1.2 Structural Eventflow Patterns

Structural eventflow patterns cover the description that conform the frame of the eventflow. That includes situations such as the appearance of sequence of related notifications (even which explicitly defined time-constraints); and relationships of causality.

5.5.2.1.2.1 Sequence Eventflow Pattern

The sequence eventflow pattern (depicted in section 4.4.3.1.2.3 Sequence Eventflow Pattern) represents the appearance of one to several events notifications after the appearance of another. In order to detect this situation, the event detector should create an abstract state machine and carry out the following actions:

- **Creation of the abstract state machine.** The creation of the abstract state machine to detect a sequence of events involves the following:
  - **Creation of the functions of the signature.** We should create functions to store the initial notification (previous event in Figure 5-32), the in functions to store each of the time variables (time...time in Figure 5-32), and finally all the in functions to store each of the events that happen after the initial event (latter event... latter event in Figure 5-32)
  - **Creation of the transition rules.** The set of rules of the abstract state machine should formalize the behaviour presented in Figure 5-32.

![Figure 5-32 Abstract state machine for the detection of the sequence eventflow pattern.](image)

Initially we must detect the first event of the sequence. Once it has been detected, we halt the detection the amount of time contained on each of the time functions (using a timed transition); and we detected each of the later event. If all of them are detected, we reach a final state, and therefore the event described by the pattern has been discovered.
The event detector should update the timing variable. The event detector should update the `currentTime` in function of the abstract state machine.

The event detector should perform all the activities associated with the event detection automaton. We are providing a recursive definition; the detection automata depicted as doted rounded squares can be arbitrarily complex. Therefore the event detector must also carry out all the activities associated with them.

5.5.2.1.2.2 Causality Eventflow Pattern

The causality eventflow pattern (fully described in section 4.4.3.1.2.1 Causality Eventflow Pattern) represents the situation when a single event causes the happening of one or several events in the system. As we consider that caused events happen strictly after the cause event, the detection of this pattern is quite similar to the sequence one. The particularity resides in that we have to explicitly check the value of `causedBy` property of the later events.

Creation of the abstract state machine. The creation of the abstract state machine to detect that an event notification causes others translates in to the event detector performing the following activities:

- **Creation of the functions of the signature.** We should create functions to store the initial notification (previous event). This in function that holds the event would be updated as soon as the event detector receives a notification of the corresponding event to be detected. Initially is set to the default value, i.e. `undef`.

- **Creation of the transition rules.** The set of rules of the abstract state machine should formalize the behaviour presented in Figure 5-33

![Figure 5-33 Abstract state machine for the detection of the causality eventflow pattern.](image)

The event detector should perform all the activities associated with the event detection automaton. We are providing a recursive definition; the detection automaton depicted as doted rounded square can be arbitrarily complex. Therefore the event detector must also carry out all the activities associated with it.
5.5.2.1.3 Iteration Eventflow Pattern

The iteration eventflow pattern (depicted in section 4.4.3.1.2.2 Iteration Eventflow Pattern) describes situations where certain happening occurs regularly or certain number of times in the eventflow. In order to detect this pattern we have opted to restrict ourselves to the case were we know beforehand the number of iterations that will happen.

- **Creation of the abstract state machine.** The creation of the abstract state machine to detect a sequence of events involves the following:
  - **Creation of the functions of the signature.** We should create functions to store the number of iterations that we wish to detect (i.e. the in function $N$) and the variable that counts the number of iterations detected (i.e. the monitored function $i$).
  - **Creation of the transition rules.** The set of rules of the abstract state machine should formalize the behaviour presented in Figure 5-34.

![Figure 5-34 Abstract state machine for the detection of the iteration eventflow pattern.](image)

- **The event detector should perform all the activities associated with the event detection automaton.** We are providing a recursive definition; the detection automaton depicted as doted rounded square can be arbitrarily complex. Therefore the event detector must also carry out all the activities associated with it.

5.5.2.1.4 Logical Eventflow Patterns

Logical eventflow patterns represent the spatial coincidence of several events in a system at the same time, or the alternative.
5.5.2.1.4.1 Conjunction/Disjunction Eventflow Pattern

Conjunction/Disjunction eventflow patterns (both described in section 5.5.2.1.4 Logical Eventflow Patterns) depict event expressions where several events happen at the same time, or situations where at least one of several events appears in the eventflow. To detect such events we believe necessary the following actions:

- **Creation of the abstract state machine.** The creation of the abstract state machine to detect either the conjunction or disjunction of several event events involves the following:
  - **Creation of the functions of the signature.** We should create a set of in functions to store the each of the events notification (event_i…event_j) in which we are interested (initially set to the default value *undef*); and a t_{out} parameter (and therefore an in function) that specifies the difference of time admissible to consider several notification as happening at the same time, plus the time expend in its detection.
  - **Creation of the transition rules.** The set of rules of the abstract state machine should formalize the behaviour presented in Figure 5-35

![Figure 5-35 Abstract state machine for the detection of the conjunction/disjunction eventflow patterns.](image)

The idea is basically the same in both abstract state machines. We give an amount of time in order to detect the notifications of the events to analyse. After that time we check if all the notifications have been detected, in the case of the conjunction pattern, or if at least one of them has occurred, in the case of the conjunction.
5.5.2.1.4.2 Negation Eventflow Pattern

The negation eventflow pattern (described in the section 4.4.3.1.3.2 Negation Eventflow Pattern) describes the situation where certain event is described as the absence of certain event in the eventflow. In order to implement the detection of this event we assume that once certain interval of time has passed, the negation of a given event expression can be considered as detected (we denote this time $T_{\text{interval}}$).

- **Creation of the abstract state machine.** The creation of the abstract state machine to detect that a given event has not happened is the following:
  
  - **Creation of the functions of the signature.** We should create functions to store whether the expression has been found or not (the input function event in Figure 5-36); and an in function $T_{\text{interval}}$ with the value of the time that the detector should observe the eventflow. These functions are kept up to date by the event detector, and initially are set to the default value, i.e. undef.
  
  - **Creation of the transition rules.** The set of rules of the abstract state machine should formalize the behaviour presented in Figure 5-36.

![Image](image.png)

Figure 5-36 Abstract state machine for the detection of the negation eventflow patterns.

As shown in Figure 5-36, this abstract state machine checks whether event that should not appear in the eventflow does. If in an interval of time does not appear ($T_{\text{interval}}$), we consider that the pattern appears in the eventflow and we end in a final state.

- **The event detector should perform all the activities associated with the event detection automaton.** We are providing a recursive definition; the detection automaton depicted as doted rounded square can be arbitrarily complex. Therefore the event detector must also carry out all the activities associated with it.
5.5.2.2 Construction of the transition rules

In order to describe the abstract state machines transition rules we specified the behaviour that the abstract state machine should exhibit in the form of automata. Let us now provide a systematic way of translating those automata into abstract state machines transition rules. This translation is necessary in order to define in an interpretable and univocal manner the operational semantics of afore pictured automata.

The core rule for each of the detection abstract state machines is common to all of them, the DETECTION_AUTOMATON rule, and is the following:

\[
\text{DETECTION AUTOMATON} = \\
\text{choice state with state } \in \text{activeStates} \\
do \\
\text{TRANSITION(state)} \\
end
\]

Basically what this rule does is to select in a non-deterministic fashion one of the active automaton states (represented as the shared function activeStates that returns a set). Once selected the state, it invokes the rule that determines, depending on the type of the automaton state, the effect of the transition on the abstract state machine (represented here using the TRANSITION rule). More specifically:

\[
\text{TRANSITION(state)} = \\
\text{if timed(state) then} \\
\text{TIMED_TRANSITION(state)} \\
\text{elsif guarded(state) then} \\
\text{GUARDED_TRANSITION(state)} \\
\text{elsif unguarded(state) then} \\
\text{UNGUARDED_TRANSITION(state)} \\
\text{elsif final(state) then} \\
\text{FINAL_TRANSITION(state)} \\
\text{endif}
\]

For each type of possible transition from a given state, a relater function is defined, namely timed : STATE → Boolean, guarded : STATE → Boolean, unguarded : STATE → Boolean, and final : STATE → Boolean. That means that when creating the abstract state machine from the automaton, we should first of all define these functions that categorize the different states.

Let us now define how we define each of these rules, TIMED_TRANSITION, GUARDED_TRANSITION, UNGUARDED_TRANSITION, and FINAL_TRANSITION.
Guarded transitions rule. In the case of the transitions on the case of guarded states, we define a rule for each state. We do so in order to include arbitrary complex condition guards just rewriting the general rule. This rule determines the next state of abstract state machine in the case of the guarded transition, and basically contemplates three possible situations.

- The first case corresponds to situations where the guard condition holds (we represent it as the guardCondition, when creating the abstract state machine it must substituted by the correspondent Boolean condition). In such case, we fire the rule GUARDED_TRANSITION_MAIN that takes out the evaluated state from the set of active states; and we add those returned by the next : STATE → {STATE} function.

- The second case appears when the guard condition is not true, but alternative states to transit has been defined for that guarded state. In such case, the actions to be taken are similar to the first case, albeit now the states to transit are specified by the
**alternativeNext**: \(\text{STATE} \rightarrow \{\text{STATE}\}\) function. These actions are contained in the \textit{GUARDED_TRANSITION_ALTERNATIVE} rule.

- Finally, if the guard condition does not hold, and the state has no alternative state to transit (i.e. \(\text{alternativeNext}(\text{state}) = \emptyset\)), we should conclude that we do not detect the complex event in the sequence of notifications. Therefore the detection must be halted without reaching a final state, which translates in the expression \(\text{activeStates} := \emptyset\). The detection will halt, since there are no pending active states to reach. This situation is also handled inside the \textit{GUARDED_TRANSITION_ALTERNATIVE} rule.

\[
\text{TIMED_TRANSITION}(\text{state}) = \\
\text{if } \text{time}(\text{state}) = \text{undef} \text{ then} \\
\text{time}(\text{state}) := \text{currentTime} \\
\text{endif} \\
\text{choice nextState with nextState} \in \text{next}(\text{state}) \\
\text{if } \text{currentTime} - \text{time}(\text{state}) \geq \text{permanenceTime}(\text{state}, \text{nextState}) \text{ then} \\
\text{activeStates} := \text{activeStates} \cup \{\text{nextStep}\} \\
\text{endif} \\
\text{if } \forall a \in \text{next}(\text{state}) \Rightarrow a \in \text{activeStates} \text{ then} \\
\text{time}(\text{state}) := \text{void} \\
\text{activeStates} := \text{activeStates} \setminus \{\text{state}\} \\
\text{endif}
\]

- **Timed transition rule.** The timed transition rule must initially check whether it has been defined some value for the function \(\text{time} : \text{STATE} \rightarrow \text{Integer}\), which stores the point in time when we first time that we reach that state.

- If it has not been defined (i.e. its location stores the \text{undef} value), we assign it the value of the global function \(\text{currentTime}\) that the event detector keeps updated.

- Then we check whether the difference between the first time that we reach that state and the current one is greater that the value specified in the time guard. For each of the reachable states from that state a different value may have been defined. Therefore the function that determines this value, \(\text{permanenceTime} : \text{STATE} \times \text{STATE} \rightarrow \text{Integer}\), has two input state parameters.

\[
\text{UNGUARDED_TRANSITION}(\text{state}) = \\
\text{activeStates} := \text{activeStates} \setminus \{\text{state}\} \\
\text{activeStates} := \text{activeStates} \cup \text{next}(\text{state})
\]
o **Unguarded transition rule.** The unguarded transition rule is rather simple. The current state becomes inactive and we transit towards the states defined by the $next: STATE \rightarrow \{STATE\}$ function.

o **Final transition rule.** Last but not least, the final transition rule is the one invoked when we reach a final state in the automaton, which in this specific case means that a complex event has been detected. The final transition must perform the following actions:

- If there is a variable associated with the detection automata (see for example Figure 5-36, the variable over the detection automaton) the final transition rule must create a fresh instance of the detected notification and assign it its basic attributes (e.g. type, timestamp, etc.). Finally the new notification is assigned it to the associated variable.

- If in the event model we define that the complex event signifies certain facts, a set of update rules should update the knowledge base of the event detector accordingly. This is one of the advantages of the use of abstract state machines. They provide both a way to define the steps for detecting a given event plus a mechanism to update accordingly the detector knowledge base.
Chapter 6  Evaluation

This chapter is devoted to the evaluation of the presented work. We propose two different evaluation procedures made to measure and validate the contributions of this thesis that depend on the nature of the work being evaluated. For the Firenze Framework we depict its usage for describing the different facets of Web and Grid services in the research projects where it was developed and reutilized. Vega evaluation also includes an empirical test to measure how the routing algorithm proposes behaves.

Finally, after presenting our work it is time to evaluate to which extent we have succeeded and achieved the objectives that we enumerated in Chapter 3 Work Objectives; checking particularly whether our work hypotheses still hold. We are going to perform two different evaluation procedures to measure and validate the contributions of this thesis. We propose two different ways due to the different nature and circumstances of the work being evaluated. For the Firenze Framework we depict its usage in different European research projects where it has been created and used for the description of the different facets of Web and Grid services, and Virtual Organizations; as well as the enumeration of the peer reviewed conferences and journals where it has been published. The evaluation of Vega, the proposed event-driven architecture, apart from theoretical considerations also implies an empirical test of the properties of the imperfect expertise-based routing algorithm.

6.1 Firenze Framework Evaluation

As we stated in Chapter 3 Work Objectives in the section 3.1 Goals, the goal of the description framework is to come up with a knowledge level description framework rich enough to express the semantics required for describing services (both semantic Web services and semantic Grid services) and Virtual Organizations. It is hard to estimate whether we have achieved this objective in a quantitative fashion. Nevertheless, we believe that a good way to evaluate the description framework is to depict how it has been used and evolved in different projects. The framework has been used in three different European Union funded projects,
Evaluation

namely Esperonto, OntoGrid, and SOA4All; with the purpose of representing different facets of Web and Grid services, and Virtual Organizations. We also believe that for demonstrating the validity of a theoretical framework it is vital to demonstrate that it has been peer reviewed. That means that, on the one hand, other experts in our field have reviewed it, but more importantly, the feedback that they have provided has been taken into account to improve the framework. This is why we also include, where possible, the peer reviewed publication of the framework.

![Diagram of the Firenze Framework]

Figure 6-1 The different uses across time of the Firenze Framework.

In Figure 6-1 we represent a summary of the uses of the framework in different projects, along with the subset of models that were used. In the following sections we depict these uses. Initially we describe those uses closer to the purpose of the framework (6.1.1 Annotation and Description of Services). Later on we describe two uses of just part of the framework, concretely of the Problem-Solving Methods model (6.1.2 Other Uses of the Firenze Framework). Both uses take advantage of the proven versatility of Problem-Solving Methods, one uses them for representing abstract provenance processes; and the other uses the possibility of gradually reify such abstract processes into more concrete ones.

6.1.1 Annotation and Description of Services

The first objective of the Firenze Framework is to describe services, both Web services and Grid services. And it is important that the later activity implies that the framework is able to describe Virtual Organizations. As we described in section 4.4 Virtual Organization Model, a Grid service description on its own does not make sense, but must be put on the context of the Virtual Organization were it is hosted. Grid services are not isolated components, but interfaces that virtualize shared resources in an institution created with certain goals.
In the next subsections, we depict the initial use of the description framework (6.1.1.1 Annotation and Description of Web Services). After that, we describe the use of the framework for the description and annotation of pre-existing of Grid services (6.1.1.2 Annotation and Description of Grid Services).

### 6.1.1.1 Annotation and Description of Web Services

In the context of the Esperonto Project the first version of the framework was developed, and initially named ODESWS (Gómez-Pérez et al., 2004a). The Esperonto Project (IST-2001-34373) was a project funded by the European Commission which aim was to bridge the gap between the current World Wide Web and the semantic Web by providing a service to “upgrade” existing Web content to semantic Web content. Esperonto Project also took into account dynamic Web content, and that included Web services.

Initially the framework consisted just in the lower models of the Firenze Framework stack. Though not as exhaustively described as it is in this dissertation, we can say that most of the top-level elements of these models were defined in this first attempt.

The framework was used to describe Web services in the following way:

- First the knowledge expert creates the knowledge level description (step 1 in Figure 6-2) that was made in a graphical and aided fashion. The knowledge expert created the description of services defined as WSDL documents.

![Figure 6-2 ODESWS description process.](image-url)
When the knowledge level representation description of the service was created, we translated it automatically to OWL-S (step 2 in Figure 6-2). The WSDL description of the service became part of the OWL-S representation, becoming its grounding.

Once the symbol level description was created, it was stored in the WebODE ontological engineering workbench. Therefore, external agents that wished to discover and use it using WebODE and its different languages translators could access the generated description.

This first version of the framework was presented in the AAAI Spring Symposium Series on Semantic Web services (Gómez-Pérez et al., 2004c). This first version of the framework was reified and later published in IEEE Intelligent Systems journal (Gómez-Pérez et al., 2004a). Moreover, a publication that described the framework plus its associated software was submitted and accepted in the European Conference on Web Services (ECOWS04)(Gómez-Pérez et al., 2004b); and demoed in the 3rd International Semantic Web Conference (ISWC2004). Finally the framework was also submitted and accepted in the W3C Workshop of Frameworks for Semantic Web Services Descriptions.

6.1.1.2 Annotation and Description of Grid Services

In the context of the OntoGrid project the framework evolved, and became ODESGS (Goble et al., 2005b). OntoGrid (FP6-511513) was a project funded by the European Commission that addressed the challenge of producing the technological infrastructure for the rapid prototyping and development of knowledge-intensive distributed open services for the Semantic Grid.

The most noticeable extensions made to the framework were the following:

- **The Virtual Organization model was created and included.** As can be seen in Figure 6-3, it was placed on top of the other layers of the model stack. This model contained most of the elements that were described in Chapter 4 Firenze Framework.

- **The Service Model was updated.** Apart from the inclusion of the formal description of Virtual Organization that we believe a must to describe Grid services, the Service Model (see 4.3 Services Model) was updated in order to be expressive enough to model Grid services. Among the necessary updates, we moved from transactional-like service description model, to describe services as reactive systems. The first incarnation of the framework considered services just as functional operations described using Problem-Solving Methods. Moreover, we raised the level of complexity of interaction

---

descriptions, in order to include several participants and more complex interaction patterns (see 4.3.2.2 Interaction Process).

- **The annotation process was enhanced.** In the context of OntoGrid the S-OGSA (Corcho et al., 2006) architecture was defined. Once of its main contributions, the concept of semantic binding, was included in the annotation process. If you compare Figure 6-2 and Figure 6-3, you notice that this inclusion changes the annotation process, enabling a level of indirection that further decouples the knowledge level descriptions made with the Firenze Framework with their symbol level translations and annotated individuals.

![Figure 6-3 ODESGS description and annotation process.](image)

The Firenze Environment (see section 4.5 Firenze Framework Design and Annotation Process) process of describing and annotating services is a generalization of the one defined in ODESGS (portrayed in Figure 6-3).

- The process begins with the creation of the knowledge level description using the Firenze Framework (step 1 in Figure 6-3). Once this knowledge level description of the entities that compose the Virtual Organization is created, it must be related with the entities that we wish to annotate (step 2 in Figure 6-3). In the case of ODESGS a WSRF service.
Evaluation

Once we have related the WSRF service with its corresponding knowledge level description, it was automatically translated into RDF(S) (step 3 in Figure 6-3). The target symbol level service ontology that we used was a simplification of the one proposed in myGRID (which can be considered a simplified version of the OWL-S ontology implemented in RDF(S)). The knowledge and representational level descriptions were both attached to the WSRF service using bindings. Knowledge bindings when associating the WSRF service with its knowledge level representation; and semantic bindings (Corcho et al., 2006) to relate it with its symbol level RDF(S) representation. Semantic bindings are created automatically from the knowledge level ones translating them to in RDF(S) as described in (Corcho et al., 2006).

Finally, all the RDF(S) representations were stored in the Atlas system. Atlas (Miliaraki et al., 2007) is a peer-to-peer system for the distributed storage and querying of RDF(S), which provides both expressive semantics-based data models and query languages; and the peer-to-peer systems properties (full distribution, high-performance, scalability, robustness, adaptability, etc.). Atlas, therefore, provides the necessary mechanisms to the later discovery of these services in a scalable fashion.

The first draft of the framework, updated with the Grid specific new additions, was published first in the 3rd International Conference on Knowledge Capture (K-CAP 2005) as a short paper. An extended version of this publication, which outlined the main submodels of the Virtual Organization Model, was submitted and accepted in the International Conference on Service Oriented Computing (ICSOC05), becoming the main reference of ODESGS (i.e. (Goble et al., 2005b)).

A more detailed description of the annotation process (along with its associated software) was described in a poster and a short paper (Gómez-Pérez and González-Cabero, 2006). The later received the 15th International World Wide Web Conference Best Poster Paper award.

6.1.2 Other Uses of the Firenze Framework

Apart from the main intended use of the Firenze Framework, parts of its model stack have been used to describe other facets of services. As we described in the introductory part of the of the chapter devoted to the Firenze Framework, we have developed its constituent models on top of each other for allowing the knowledge expert to pick just those that she need.

In the following sections we depict two use cases of the Problem-Solving Method model of the Firenze Framework, one for interpreting past executions of services (6.1.2.1 Firenze
Framework and KOPE); and the other to compose new services expressed in the form of processes (6.1.2.2 Firenze Framework and the Automatic Composition of Web Services).

### 6.1.2.1 Firenze Framework and KOPE

In the context of OntoGrid, independently to the work presented in this dissertation, KOPE (Gómez-Pérez and Corcho, 2008) (Knowledge-Oriented Provenance Environment) was developed. KOPE is a stand-alone system that could be used to analyze provenance logs. By the time that KOPE was developed, widely adopted provenance descriptions initiatives such as (Moureau et al., 2010)(Moureau et al., 2011) and the work of the W3C Provenance Working Group were not still available. Moreover, KOPE required a formalism that allowed different levels of abstraction for interpreting provenance information. The choice was a metamodel of Problem-Solving Methods constructs organized as library with a hierarchy of methods and instances of the Problem-Solving Methods; and the set of domain ontologies that describe the application domain. For representing the Problem-Solving Methods of the library, KOPE used the Firenze Framework Problem-Solving Methods Model.

![Figure 6-4 Using Problem-Solving Methods for interpreting logs.](image)

In Figure 6-4 we depict the activity that KOPE performs. KOPE engine takes afore enumerated knowledge elements, and identifies these tasks in the provenance store’s process documentation. Therefore, it provides an abstract user-oriented view of the low-level provenance log. As described in (Gómez-Pérez and Corcho, 2008), the KOPE engine detects

---

29 [www.w3.org/2011/prov/wiki/](http://www.w3.org/2011/prov/wiki/)
task occurrences via twig join algorithms, (Bruno et al., 2002)(Chen et al., 2005), performing a matching between the process enactment documentation and each Problem-Solving Method knowledge flow. As it is also depicted in Figure 6-4, the matching is performed at each abstraction level ($level \ 1$, $level \ 2$, etc. in Figure 6-4), providing thus a multilayered interpretation of the provenance log.

KOPE using the Firenze Framework as its description framework was presented in the Second Provenance Challenge$^{30}$ and it is fully described in (Gómez-Pérez and Corcho, 2008) and (Gómez-Pérez, 2009).

### 6.1.2.2 Firenze Framework and the Automatic Composition of Web Services

SOA4All (FP7-215219) was a project funded by the European Seventh Framework Programme, which motto is to realize a world where billions of parties are exposing and consuming services via advanced Web technology. In other words, its objective is to provide a comprehensive framework that integrates complementary and evolutionary technical advances (i.e., service-oriented architecture, Web principles, Web 2.0 and semantic technologies) to bring service orientation outside of the boundaries of big companies.

In the context of the SOA4All$^{31}$ project part the Firenze Framework stack was used for the automatic composition and adaptation of processes; more concretely the Problem-Solving Methods Model.

SOA4All processes, as the project restricts to service-oriented environment, are automated and executed by services (i.e. services become their building blocks). These processes and its constituent services must be adapted to particular needs in a particular context; being configured on a per case basis.

---


$^{31}$ [http://www.soa4all.eu](http://www.soa4all.eu)
The SOA4All approach lets users handle abstract and easy to use (and understand) processes representation, whereas the system performs later a context-dependent customization transparently. Therefore, the user handles a simple process composed of coarse-grained tasks, and the system carries out intelligently the transformation into a complex yet concrete and executable process\textsuperscript{32}. This transformation is made in terms of the execution context, the current state of affairs at design time and very simplified user requirements.

The transformation from abstract to executable processes is performed by means of a knowledge-intensive configuration process, a parametric design process (the big arrow in Figure 6-5). Parametric design is a simplification of the design as configuration procedure. Configuration-based design is a simplified form of design where a set of pre-defined components\textsuperscript{33} is a priori selected and combined till it satisfies a set of requirements and obeys a set of constraints (Mittal and Frayman, 1989). Parametric design further refines configuration-based design, assuming the existence of functional solution templates that guide the design process, decreasing greatly the complexity of the design process. Parametric design restricts thus the space of possible designs, assuming that the targeted artefact can be expressed in the

\textsuperscript{32}As concrete executable process, as depicted in Figure 6-5, we refer to those processes composed only by concrete tasks; by concrete tasks we refer to those that are realized by a primitive method and therefore represent a single step of computation (i.e. the invocation of a single service operation).

\textsuperscript{33}The set of components considered for a design as configuration task receives is denoted as technology. We remit the reader to (Wielinga and Schreiber, 1997) for a clear description in all these matters.
form of parameterized functional templates. These parameterized functional templates, in our case, are expressed using Problem-solving Methods and their facility to express processes in different levels of abstraction.

Parametric design can be characterized as search (we recommend (Motta, 1998) for a deep description of this topic). This process represents a search in a space of subassemblies (Chandrasekaran, 1990). From an initial design (depicted as a red circle denoted design plan in the literature, and in our case the abstract process that the user handles) we navigate thorough the space of possible intermediate design structures till we reach a design that we consider a final solution (an executable process with certain characteristics in the case we are in).

As we show also in Figure 6-6, in order to increase the scalability of this process, we extend the classical approach to this synthesis task proposing a multi-agent opportunistic approach. Designs are stored on a blackboard; agents consult that blackboard and propose new designs. Nevertheless the concept of search remains the same, with the difference that in this case each agent posses its own heuristics (i.e. each agents posses its own set of Problem-Solving Methods); and as a consequence, each agent explores only a subpart of the search space in a parallel fashion with respect of the rest.
In our concrete problem, each design corresponds to a high-level process expressed as a Problem-Solving Method, plus a set of design constraints and requirements. Design operators modify this high-level process, depending on the set of constraints and requirements.

The main design operator is the substitution design operator. Using the assumptions about the context contained in adapters, an agent selects which process template fits better in the actual design context (i.e. makes true all constraints and requirements). In Figure 6-7 we show how a composite task is related with different processes templates. Let us suppose that there is a design placed in the blackboard which high-level process contains Task. The substitution design operator, using the assumptions and applicability conditions contained in the adapters (adapter$_i$ and adapter$_a$) plus the actual context, chooses the most appropriate process template to replace Task. These processes templates correspond with the controlflow of the methods that solve Task, (Method$_j$ and Method$_i$ in Figure 6-7). Finally the agent substitutes all the appearances of Task with the chosen process templates and places the new design in the blackboard.

Once an agent has generated a final design (i.e. a process which tasks are all atomic and that satisfies all the requirements), it places it in the blackboard. Other type of agents, the validation agents, check if it is a final result, and in such case they notify and the resulting design is returned to the user.

This composition environment has been applied in two of the SOA4All use cases, namely an e-government and a telco 2.0 use cases. A paper describing this approach, i.e. (Lecue et al., 2010), was published in the research track of the 8th IEEE International Conference on Web Services (ICWS 2010).
6.1.3 Conclusions

To conclude and as a summary of the evaluation of the Firenze Framework let us revisit the initial work hypotheses that we introduced in Chapter 3 Work Objectives.

### Hypothesis 1
The use of knowledge level framework for the formal description and design of service-oriented environments enhances the quality of the design, as design and implementation phases are kept separated. In concrete, we foresee that it fosters the design reusability.

As we have depicted in the different uses of the Firenze Framework, the knowledge expert can focus in the creation of the high-level design of the services, whilst the creation of the concrete implementations of the service is done automatically. In consequence, there are no design decisions made in terms of the characteristics of the target symbol level representation. Moreover, the reusability of the models made with the Firenze Framework are guaranteed in two fashions:

- **One design, several representations.** The description of any of the entities described in the Firenze Framework description (e.g. service, role, agent, etc) may have associated different symbol level representations in different representational languages and formalisms; and with different detail granularities. In the afore described uses of the framework in research projects for example, a single service description can be automatically translated to two different symbol level representations: a RDF(S) implemented service ontology, and OWL-S.

- **One design, several domains.** All the service operations descriptions, since we use Problem-Solving Methods, are made where possible with a clear separation between the domain they act upon and the processes they carry out. As the descriptions of services, and parts of the Virtual Organization description, are in part built on top of these reusable formalizations, part of the design made with the Firenze Framework can be effortlessly reused in different domains.
Evaluation

**Hypothesis 2.** Problem-Solving Methods, abstract design patterns and ontologies are valid tools for the description of semantic Web services and semantic Grid services.

This is the use that the Firenze Framework was given in the Esperonto, OntoGrid and SOA4All projects. In Esperonto and OntoGrid we focused in those parts more related with the automatic discovery of semantic services, defining the service profile and simple problem-solving method descriptions (descriptions that appears in (Goble et al., 2005a)(Goble et al., 2005b)(Gómez-Pérez and González-Cabero, 2006). In SOA4All, as we have described, the Firenze Framework was extended and used to represent service operations processes in order to enable the automatic composition of semantic services (use that appears in (Lecue et al., 2010))

**Hypothesis 3.** Problem-Solving Methods, abstract design patterns and ontologies are valid tools for the definition of formal and explicit models for describing the different aspects of Virtual Organizations.

This was precisely the use that was given for the Firenze Framework in the OntoGrid project and was published in (Goble et al., 2005a)(Goble et al., 2005b)(Gómez-Pérez and González-Cabero, 2006).

**Hypothesis 4.** Services (especially Grid services) are better described as reactive systems rather than transformational systems.

After the creation of the Firenze Framework we should rephrase this hypothesis to Web services are better described as reactive systems than transformational systems; but not all Grid services can be formally described as transformational systems, specially those that expose non-terminating complex interfaces.

Web services frameworks can characterize services as transformational systems since target the description of application services solely (i.e. those that implement the business-level capabilities of the organization). Considering services as reactive systems brings the possibility of describing more complex services, such utility services that deal with application-agnostic functionality, such us for example job management, logging, monitoring and management
services. It is so because a reactive system description, contrary to the description of a transformational one, entitles the description of a non-terminating ongoing behaviour, a description of the effects in the environment that the system produces by enabling, enforcing, or preventing certain events. This description includes the possibility of many agent interacting at the same time with the reactive system, parallel threads of interaction and internal and external observable states specifications. And these description capabilities are necessary to describe the full spectrum of possible Grid services.

6.2 Vega Evaluation

With regard to Vega, the design principles that uphold it (that we have presented in 5.1 Vega Architectural Principles) have been peer reviewed and published as part of (Domingue et al., 2008b) in the second IEEE International Conference on Semantic Computing (ICSC 2008); and also in (Davies et al., 2009) in the British Telecom Technology Journal.

Once the design principles that we propose were peer reviewed, in order to analyze the suitability and usefulness of the architectural concepts and algorithms that we exposed in Chapter 5 Vega, we have developed a prototype infrastructure that implements them. In this section we present the most representative experiments that we have carried out to evaluate the main design characteristics that we seek: scalability and two aspects of self-manageability, namely self-configuration and self-optimization.

6.2.1 The eInclusion System Use Case

The presented experiments are defined in the context of the ongoing work that we are performing in the development of an ambient intelligence environment that would enhance the autonomy of people with different functional diversities. In this system we develop the idea of Universal Control Hub (Zimmermann and Vanderheiden, 2007), which rationale is that a person with its own and adapted controller device (e.g. mobile phone, PDA, universal controller, etc.) should be able to interact and control different devices (television, door locks, ATMs, and a long etcetera), as well as external software services. With such controller, we introduce the necessary level of indirection that allows us to introduce the needed assistive technologies to bridge the possible handicap gap between the user and the device. We refer as assistive technology to the hardware or software that is added or incorporated within a system than increases accessibility for an individual, as defined in (ISO/IEC 24756, 2009).
In our experiment we consider the beginning of the interaction of the user with such environment. As seems logical, before the user is able to perform any task on any of the target devices that conform the system, the system must provide the list of tasks that she or he is allowed, and more importantly, able to accomplish in that environment. We use the Firenze Framework to represent the services that virtualize the target devices. The construction of the list of tasks that the user might be able to carry out in the system translates in to a process of service discovery and capability matching. Most semantic services approaches would place all the descriptions of services in a huge repository that would act as a centralized knowledge base that would be queried later on synchronously by the client (the user controller device in this case). In this experiment we depict an event-driven approach to this activity. The user’s controller announces itself to the environment. This announcement contains both its profile and the users common access profile; and after that, the controller starts receiving the description of the tasks that might be of interest to the user. And more importantly for an inclusion scenario such the one we are proposing, the system only sends those tasks that the user is effectively able to use according to her or his capability profile.

This process of finding which tasks a user might perform is performed in a totally distributed fashion, as we use the event-driven facet of Vega and its intelligent event routing and handling infrastructure. What we propose is that each of the services that either virtualize a device or provide some functionality in the environment, is also responsible of announcing their capability to new controllers that might be interested.

34 Obviously this use case depicts the case of the first interaction with the system. Later on, the user will for sure have the list of its commonly used devices in its controller device and this discovery process is not longer a prerequisite to start controlling a target device.
In Figure 6-8 we depict graphically a generic service \textit{service}. As it is shown in its interaction process, each service interacts with the environment in basically two parallel ways each of them involving a different mode of interaction.

- \textbf{Reply/response model of interaction.} On the one hand, the service exposes its functionality with a set of operations \((\text{op}_1, \ldots \text{op}_j)\) that can be directly invoked by the user’s controller (portrayed as the \textit{performtask}_i, \ldots \textit{performtask}_j in the interaction process of \textit{service}).

- \textbf{Event-driven model of interaction.} On the other hand, the service interacts with the environment in an event-driven fashion, listening to notifications of new controllers. When the service receives such notifications, as formalized by the task announce capability depicted in Figure 6-9, it generates a capability announcement that is later delivered to a user device.
Thanks to the intelligent notification routing, each service is not overwhelmed by all the notifications that happen in the system. Instead they only receive those notifications that represent a combination of user and controller that are actually able to access to the target device that the service virtualizes. The drawback is that the notification system needs to handle complex service descriptions in order to perform such evaluation (Firenze Framework descriptions in our case). The domain dependant part of these service descriptions model (i.e. the eInclusion domain\(^{35}\)) involves information of different nature, what makes the description even more complex. Among the most important types of information that are:

- **Multifaceted non-functional description.** Non-functional information is especially relevant for the considered scenario. Information such as the location of the devices virtualized by services is of great importance for an ambient intelligence environment. For example, an operation such as open the third floor door, needs that the user is nearby that door. Of course, other non-functional information is also considered, such as security (e.g. the user might pose some security target such as privacy), QoS, reliability, etc.

- **Capability and modality description.** In order to determine whether the user is capable of using certain virtualized device, we must specify for each service the characteristics of the communication that will be established between the user (plus her or his controller) and the controlled device. There is a difference between being able to control a device and really being able to use it as it is meant to be. Thus, in the description of the device operations we must include information about modality and capability. More precisely:

  - The **modality information** describes the path of communication between the human and the device. We represented it as a set of media types where the communication is transmitted (e.g. written text, spoken text, tactile graphics, gestures, textures, etc.). Note also that each of these media types might have additional information associated with them, such as the media language code (expressed using the three symbol code defined in the ISO 639-3 standard (ISO 639-3, 2003)), the physical channel, the capacity of the media, etc.

  - The **capability information** is used to determine whether a user with certain functional diversity might be able to interact with this operation. Once again we will stick to the Common Accessibility Profile framework (ISO/IEC 24756, 2009), which recommends

\(^{35}\) The eInclusion domain used in this experiment is formalized using a set of ontologies were developed in the context of the INREDIS (INterfaces for RELations between Environment and people with DISabilities) Project, which is part of the Spanish government's INGENIO 2010 initiative managed by CDTI (the Industrial Technology Development Centre).
the use of the standard ISO 80000 (ISO/IEC 80000-13, 2008). Accordingly, we define a set of concepts that represent different capacities (e.g. hear, speak, touch, etc.) and its instances are associated with triplets that specify the upper value, the lower value and the unit of measure (e.g. a device might require that the user is able to hear frequencies between 64 and 65536 Hz).

6.2.2 The eInclusion System as a Vega Deployment

Once we have described the expected functionality of the system in a generic fashion, let us evaluate the concepts and algorithms that we proposed in the chapter devoted to Vega. In Figure 6-10 we portrait the simple arrangement of components that we have set up for making our experiment manageable (i.e. using this architecture we can setup arbitrarily complex deployments). It is composed by:

- **Semantic service bus.** The semantic service bus, as we described in 5.2.1 Service Bus (page 151), is composed by a set of semantic-aware intelligent event brokers that use knowledge as a medium to achieve self-management and scalability. As we depict in Figure 6-10 it is composed by an external layer of local brokers \((\text{localBroker}_A \ldots \text{localBroker}_E)\) that act as gateways to the bus which are linked to a farther inside layer of border brokers \((\text{borderBroker}_A \ldots \text{borderBroker}_B)\) and an innermost layer of brokers \((\text{innerBroker}_A \ldots \text{innerBroker}_B)\) which route notifications. The presented topology is small, we should not expect it to handle a huge number of services types; but it is handy and we will be able to configure and control the state of its components.
Evaluation

- **Controller detectors services.** On the left side of the semantic service bus we plug the controller detector services. In this simple experiment we suppose that there are five zones where the eInclusion system is deployed. On each of these zones we place a service that detects new controllers (i.e. $\text{controllerDetector}_{\text{zone0}}$…$\text{controllerDetector}_{\text{zone5}}$) and sends the correspondent notification via its assigned local broker (in this case either $\text{localBroker}_A$, $\text{localBroker}_B$ or $\text{localBroker}_C$).

- **Task provider services.** On the right side of the semantic service bus presented in Figure 6-10 we connect the generic services that virtualize devices (those illustrated in Figure 6-8). We have labelled them as task providers because in our experiment they just send a list with the tasks that describe the operations they perform\(^{36}\). We have deployed 12 service types (basically we wanted to have at least 2 for each zone to check that the routing was properly performed) with 20 service instances each. The number of instances is not as important as the number of service types, since the delivery is only made in terms of services types (this is one of the advantages of expertise based routing).

With this configuration we have performed our experiment that we divide in two clearly separated steps:

- The **first step of the experiment** is the brokers expertise creation. The creation of the expertise is described in section 5.4.2 Expertise Knowledge Representation and Creation, we have just taken the algorithms described in this section and implemented them; implementation that follows step by step the algorithm for each of the brokers type. We have not performed a detailed analysis of the time that takes to create the knowledge network overlay in this experiment. On the one hand the size of brokers network of the experiment is not big enough to (in fact for this simple scenario takes less than a second to set up all the topology); and in case that the size of the brokers network gets bigger, there would not be any problem either, since we propose an incremental algorithm. Despite this consideration, this first step of the experiment demonstrates that the configuration of Vega is carried out in a consistent manner autonomously (i.e. self-configurable). The addition of new types of services, the spawn of new services instances; and its possible removal is automatically and transparently handled and replicated in the brokers topology.

- The **second step of the experiment** corresponds, as we have already stated, to the beginning of the interaction of the eInclusion environment. This part of the experiment matches with

\(^{36}\) *Senso stricto the whole profile of the service should be sent, since non-functional properties are also sent to the user device.*
the construction of the list of tasks that the user might be able to carry out in the system. In
our experiment, as we take the eIncusion system perspective, this process begins when the
new controller detector service (i.e. service \texttt{controllerDetector}\textsubscript{zone}) detects a new controller.
The service receives the profile of both the user and the device, encapsulates this
information as a new notification of the event new controller and sends it to the semantic
service bus (i.e. delivers the notification to its associated local broker). Then we measure
how much time takes to the semantic service bus to deliver this notification to the tasks
providers of this zone, since is equivalent to the time it takes to discover this service.

We have run several experiments with this scenario using different configurations. From all
these trials we have selected two that we find most representative, namely the best-case scenario
and the worst-case scenario.

\subsection*{6.2.2.1 The Best-Case Scenario}

In the best-case scenario we send each 250 millisecond a notification to either the \texttt{localBroker}_A, \texttt{localBroker}_B or \texttt{localBroker}_C of the detection of a new controller device. In this
scenario (and those that resulted in better outcome) we find that two circumstances occur at the
same time:

- \textit{Local brokers share the same notification traffic}. The throughput of the incoming
  notifications is well distributed along the external layer of local brokers. As we send them
  randomly either to \texttt{localBroker}_A, \texttt{localBroker}_B or \texttt{localBroker}_C we can expect that more or
  less each of them will receive the same number of notifications.

- \textit{Notifications are distributed in time}. Thought we increase the number of new controllers
  notifications, the increase is done reducing the interval time between notifications, not
  sending notifications simultaneously.

These 250 ms correspond approximately to the highest notification frequency that makes this
system without self-manageable routers stable (i.e. response times are uniform). With this
frequency we have on the one hand a reference response time to compare our algorithm; and on
the other hand, we use this frequency to train the inner brokers in order to determine the value
of their expected duration. We represent the response times of the system in Figure 6-11 as the
white series.
After finding the maximum throughput that the system can handle without incurring in performance degradation, we increase the frequency till we find the highest frequency that the self-managed brokering system is able to cope with, which is almost twice than the one that the system could previously handle. We represent the timing in green, and we label it time 2x frequency (with relaxation) in Figure 6-11. If we introduce a flow of notification with this frequency in a system with no self-management, we obtain the last series represented in Figure 6-11, labelled time 2x frequency (without relaxation). As can graphically be seen the improved response time is about 60% of the response time of non self-configured. And also is important to note that the system is in a stable state and therefore the improvement is guaranteed as long as the notification flow peak last.

6.2.2.2 The Worst-Case Scenario

We turn now to the worst-case scenario that we have identified from the set of experiments that we have performed so far. After running several experiments we have found the configuration that though the requirements in terms of notification traffic are not especially high, the resulting response times are as satisfactory as we expected (despite the fact that it illustrate again the added value of the algorithms that we propose in this dissertation).
Evaluation

scenario is quite revealing and of great importance since on the one hand it permits us to validate the algorithm, and on the other hand it is very useful to determine the next steps toward the improvement proposed architecture.

In this scenario we produce 6 notifications on each zone, simultaneously. Moreover, we have on purpose assigned more zones to one of the local brokers than for each of the other brokers. Therefore, in this scenario two circumstances occur at the same time:

- **The topology is not well balanced.** We have not equally distributed each of the services that generate new notifications in the system. We have assigned to *localBroker* \(_A\) the detectors of *zone*0, *zone*1 and *zone*2 therefore exposing *localBroker* \(_A\) to too much notification traffic.

- **The notifications enter in the brokering system synchronized.** Contrary to the case that we analyzed in the previous section, though the frequency (both normal and augmented) of the detection of new user devices is similar, all the notifications happen exactly at the same time. Therefore, if in our experiment we create 6 notifications per period and all of them appear at beginning of the period (in a real-world use of this system this is highly unlikely, simultaneous events in distributed systems are almost impossible).
In the $z$-axis we represent the zone where the notification was generated (i.e., generated by the correspondent $controllerDetector_{zone}$ of the $zone$). The $x$-axis represents time (counted in samples of 1200 milliseconds); and the $y$-axis represents the time it took from the moment that a new controller is detected by a controller detector till the detector is delivered with the list of available tasks (measured in microseconds).

Figure 6-12 Worst-case scenario graphical representation.
We have determined that the system remains stable with no self-management till 6 notifications each 1200 milliseconds. If we increase the incoming traffic just a 15%, we obtain the results represented in Figure 6-12.a and Figure 6-12.b. As can bee seen, the system response times are not stable and they increase in a lineal fashion. In the case without self-management the response time of the system reaches response time peaks of up to 8 seconds in just 50 seconds functioning.

In Figure 6-13.a we display the average time for each of these time series that appear in and Figure 6-12.a and Figure 6-12.b. As we did in Figure 6-11, we have represented this series alongside the average of the non-relaxed system when it is not overloaded (6 notifications each 1200 milliseconds).

Even in this case, when we compare the self-managed brokers with relaxation approach to the conventional ones, we find a clear improvement. As shown in the graphic, for the period of time of 50 seconds in this experiment (and with such simple brokers topology), we can assure that the discovery process response time is reduced around a 35%. And this reduction is possible thanks to the self-management of inner notification brokers.
We conclude thus, that the novel imperfect expertise based delivery algorithm that weakens filters evaluation on brokers, is able to scale and sustain peaks of higher throughput of event notifications. As we have said before, the presented deployment of the architecture has been made very simple for monitoring and controlling purposes, but it has shown that it can easily handle periods where the traffic of notifications doubles the one it was provisioned. We can assure that with the addition of new inner brokers we will for sure increase the time that the system can handle an overload in notifications traffic (or even become stable despite the overload, such in the case of the best-case scenario). The cost would be just to sacrifice the minimum response time.

Finally, after analyzing the logs of the inner brokers we found what makes causes the bad results of the worst-case scenario in comparison with the other experiments (specially the best-case scenario). The problem resides in that we have clearly underestimated the impact that the overwhelmed local broker might pose to the overall system scalability. It is recommended that further research be undertaken in this topic, but using the same techniques presented so far we can outline a solution. We have provided the necessary mechanism to determine whereas a broker is overloaded, we can use it to determine when a local broker is (it can be either because it has too many describers, or because some of them generate too much traffic). Instead of relaxing the binary decision diagrams descriptions (it can break the safety condition\textsuperscript{37} of the brokering system), one possible policy could be to update its neighbourhood changing the subscription of one or various services to other local broker in a more relaxed state. We should also contemplate the possibility of having more than one local broker for those services that produce too much traffic for just one broker. This is one of the reasons that makes concept of adaptable topology the first topic in our future work list that we enumerate in the next chapter.

\textsuperscript{37} The safety condition states that we should deliver a notification to a subscriber if it matches one of the component current subscriptions (Mühl et al., 2006). Therefore, local brokers cannot use approximate delivery, since each failure would result in the delivery of a non-solicited notification to a subscriber.
6.2.3 Conclusions

To conclude, and as a summary of the Vega evaluation let us revisit the initial work hypotheses that we introduced in Chapter 3 Work Objectives.

**Hypothesis 5.** Complementing semantic service-oriented architectures with event-driven principles benefits the provisioning of knowledge-intensive services.

As we have already stated in the chapter devoted to the architectural design principles event-based systems carry the potential for easy integration of autonomous, heterogeneous components into complex systems that are easy to evolve and scale (Bates et al., 1998)(Mühl et al., 2006). The event-driven architectural style is not a replacement of the service-oriented architecture paradigm, but a complement. At a glimpse, Figure 6-14 depicts how these approaches complement having different characteristics and assuming different responsibilities. The service-oriented architecture exposes functionality in the form of services, providing functionality and mechanisms for the request and response of information. The event-driven architecture is responsible for perceiving situations of interest; and deciding which actions should be performed in accordance to these situations of interest by delivering the necessary information to the appropriate services.

Figure 6-14 Service-oriented vs event-driven architectures (adapted from (Kazi, 2005))
In consequence, we believe that combining an event-based architecture with semantic service-oriented architecture (which are a subtype of service-oriented architectures) brings the following benefits:

- **Improves reactiveness.** A service-oriented architecture complemented with an event-driven architecture gains the ability to detect events or relevance in the environment (owing to the event detection architecture); and in a matter of millisecond choose intelligently the service that should address such situation (owing to the event delivery architecture). Therefore adding complex event detection plus using an event-driven approach allows the system to diligently respond to time-sensitive situations. As it is outlined in (Kazi, 2005) the key for truly service-oriented real-time system is to quickly sense and respond to changes, threats and opportunities; changes, threats and opportunities that manifest themselves as event notifications.

- **Improves scalability** (reliability and performance). Adding an event-driven facet to service-oriented architectures improves scalability. The key enablers for such improvements are rooted in its interaction model. The first important advantage is the lack of interaction season (what in (Hohpe, 2006) is referred as the lack of call stack). Event-driven systems are asynchronous in nature, agents do not wait for the receiving system to process events, instead they can continue carrying out other activities. In such systems there is no server that has to maintain interaction sessions with clients. The second great scalability enabler is how easily event-driven architectures introduce one to many communications. As depicted in Figure 6-14 service-oriented architectures propose a request/response model that fixes both ends of the communication (i.e. the interaction entitles one consumer and one producer). To the contrary, in the event-driven interaction model neither the published notifications nor the subscriptions are directed toward specific components, remaining the number of recipients open. This fact, along with the low coupling among services, easily allows introducing replication of services. In the concrete case of Vega, due to the intelligent notification delivery (and its self-configurable nature); brokers adapt themselves to situations where notification traffic overwhelmed them, what promotes scalability. In cases in which the relaxation mechanism is not enough, it is easy to adapt at run-time the event brokering architecture just changing the subscription of the most resource demanding services to those local brokers which are in an underused state (we left the automatization of such procedure as a future line of our work, as we shall describe in Chapter 7 Conclusions and Future Lines)
**Evaluation**

**Hypothesis 6.** Expertise-based delivery algorithms reduce the size of knowledge associated with the routing activity.

In the definition of the topology and evaluation of the proposed routing algorithm the number of entries roof for a given broker is the size of its neighbourhood. This is precisely why we choose to adopt the SWAP expertise approach. Having such a small and fixed entry set size is feasible from a computational point of view thanks to the use of the relaxed expertise descriptions and the imperfect notifications routing algorithm.

**Hypothesis 7.** Allowing a certain degree of relaxation of the safety delivery condition, inner event brokers might not become overwhelmed resulting in an increase of the overall system scalability.

As we have shown in the experimental evaluation, the brokering architecture is able to scale and sustain peaks of higher throughput of event notifications owing to the novel imperfect expertise based delivery algorithm that weakens filters evaluation on brokers, As we have said before, the presented deployment of Vega is very simple for monitoring and controlling purposes, but it has shown that it can easily handle periods where the traffic of notifications doubles the one it was provisioned. With the addition of new inner brokers, we would observe an increase of the system minimum response time (since we increase the number of hops that a notifications should take to arrive to the suitable destinations); but the brokering system would be able be able to handle temporarily high overloads of notifications traffic (or even become stable despite the overload, such in the depicted best-case scenario).
Chapter 7  Conclusions and Future Lines

This chapter presents the conclusions of our work, focusing on the main advances to the state of the art on service description frameworks and semantic services provisioning architectures. In this chapter, we also identify open research issues in both areas that will delineate our future lines in both the description framework and the event-driven architecture.

In this last chapter we present the conclusions and future lines of the presented work. The conclusions are composed of a summary of the addressed open research issues, along with a digest of the main advances on the state of the art of service description frameworks and semantic services provisioning architectures. Finally, we include a non-exhaustive list of the main extensions of the showcased framework and architecture that we plan to carry out in the near future.

7.1 Conclusions

As described in Chapter 3 Work Objectives, the main objectives of the work presented in this thesis are to advance the state of the art on service description frameworks and service provisioning architectures. Our main objectives have been:

- A knowledge level description framework rich enough to express the semantics required for describing services (both semantic Web services and semantic Grid services) and Virtual Organizations.

- An event-driven service-oriented architecture for the semantic Web and semantic Grid that using the knowledge level description framework, and by means of its self-management capability, is able to achieve high-scale provisioning of semantic services.
Conclusions and Future Lines

7.1.1 Summary of the Addressed Open Research Problems

Several open research problems have been taken into account in order to achieve the enumerated goals. In order to achieve the first objective, the knowledge level description framework, the following list of open research problems were to be addressed:

- **A Grid specific framework.** Though when we started the presented work there were several description frameworks for describing Web services in a formal fashion, none of them could be directly used to describe Grid services. The differences between Grid services and Web services made mandatory the creation of a richer and Grid specific description framework.

- **Annotation methods and techniques.** From a methodological perspective, there was a lack of integrated methods and techniques that support annotation and design of semantic services.

- **A high-level description framework.** Service description initiatives, though quite complete such the case of WSMO, broke the separation between the design and implementation phases, working at the symbol level. This fact affected considerably designs quality and reusability. Consequently we believed necessary the creation of a description framework that would allow the creation of high-level and reusable designs.

The second objective, the event-driven service-oriented architecture, unveiled also new open research problems that we could not neglect. The goal was the definition of a truly distributed knowledge-based architecture, able to achieve scalable semantic services provisioning. Its main characteristics were its inherent distributed nature and the change of mode of interaction. In order to achieve these goals, we indentified the following unsolved issues that should be tackled:

- **A scalable notification service.** From a non-functional point of view, design characteristics like scalability, efficiency, availability, etc are heavily influenced by the selection of the notification service architecture. In order to create such a scalable knowledge-intensive notification service, we realized that we should solve first the following subtopics.

  - **Peer-to-Peer knowledge-based routing.** We needed to define a decentralized system that using a rich shared events model, should be able to route event notifications on the basis solely of the brokers expertise.

  - **Increasing scalability by preventing brokers overloading.** The problem of using complex services description for routing notification was the overload introduced when handling these descriptions. This overload seriously affected the overall system
Conclusions and Future Lines

scalability, as notification brokers might become overwhelmed when routing notification on the basis of such descriptions. In order to make the brokers more diligent and prevent this situation, we identified two key open research issues,

• **Routing knowledge oversize prevention.** Knowledge about routing size increases until becomes unmanageable; especially if we pick a broker located in the innermost parts of the routing infrastructure. We needed an approach capable of creating compact routing data structures in the case of complex advertisements and subscriptions expressions.

• **Routing complexity adaptability.** Innermost brokers of the brokering infrastructure sustain a much heavier notification traffic that those situated toward the edges. An adaptable algorithm with the ability to work in different levels of detail was a open research problem when we started defining the proposed architecture.

  o **An autonomic self-configurable system.** Large systems, or small but dynamic ones, cannot rely on human administration, since the system extensibility and reactivity might be affected. Unlike present semantic services architectures; we believe necessary the achievement of one of the autonomic computing attributes, self-configuration, and the self-optimizing attribute where possible.

7.1.2 Summary of the State of the Art Contributions

Let us summarize the contributions to the state of the art direct result of the work presented in this thesis. Regarding the first goal, the Firenze Framework, we consider that we have obtained the following outcomes:

  o **The Firenze Framework is a Grid specific description framework** capable of describing Grid entities. The key features and novelties of this facet of the description framework are the following:

    • **Virtual Organizations description.** We have created a description framework that provides a formal and explicit description of this type of virtual institution, which goes beyond the simple sum of the descriptions of its services; specifying both the rules that govern the interaction between the entities involved in the Virtual Organization; and the entities themselves.

    • **Provenance.** The Firenze Framework can be used to provide the formal specification of the origin and metadata information of a concrete enactment of a Grid service, as is presented in (Gómez-Pérez and Corcho, 2008).
Conclusions and Future Lines

• **Complex interactions.** The Firenze Framework is able to describe complex interactions involving multiple parties playing different roles, multiple stages, and multiple views. Moreover the exchanged messages can also be arbitrarily complex.

• **Reactive systems and Grid services instances.** The Firenze Framework, contrary to all the pre-existing service description frameworks, describes service instances as reactive systems. Therefore the framework is able to describe the functionality, lifecycle and external/internal states of a concrete instance of a Grid Service.

  o **The Firenze Framework is a knowledge level description framework.** From this perspective its contributions to the state of the art are the following:

    • **Guidance in knowledge acquisition.** The use of a model in terms of knowledge is equivalent to knowledge acquisition; the proposed model itself can be used as a frame for this knowledge acquisition process.

    • **High-level functional specification and design of the system.** As a knowledge level framework the Firenze Framework clears up the misunderstanding between knowledge and representation providing:

      − **A high-level design of system.** The use of a high-level framework is less error prone. Moreover the quality of the design is not constrained by the characteristics of the target symbol-level representation language.

      − **Sharing and reuse.** From a single design we may obtain different representations, in different languages, expressivities, granularities, and with different reasoning mechanisms.

With respect our contribution to the state of the art, of semantic services provisioning architectures in this case, we have defined and implemented Vega, an event-driven scalable semantic services provisioning architecture that has the ability to self-configure. More precisely:

  o **Self-configurable architecture.** The configuration of Vega is carried out in a consistent manner autonomously (i.e. self-configurable). The addition of new types of services, the spawn of new services instances; and its possible removal is automatically and transparently handled.

  o **Scalable architecture.** Vega is able to handle a huge number of annotated services instances, a number that may vary across time quickly. This is possible owing to two key contributions of the presented architecture to the state of the art.

• **Change in the interaction model.** Vega is the first semantic service-oriented architecture that changes the model of interaction from the request/response model to an
Conclusions and Future Lines

event-driven interaction model. From this perspective, this architecture can be considered as complementary to the already existing semantic service provisioning architectures.

• **A novel imperfect expertise-based delivery algorithm.** Such algorithm weakens filters in the evaluation on brokers that sustain a high throughput of event notifications (as in the case of the bus innermost brokers). This enables a decentralized peer-to-peer broker network that uses knowledge about the components to route event notifications.

7.2 Future Lines

Even though we believe that this thesis presents important contributions, there are still some related open research problems that have not been addressed. Some of them, which appear as restrictions in the section 3.4 Work Assumptions, Hypotheses and Restrictions have been intentionally left aside to make the targeted objectives less daunting. Others have appeared while its development or they are even a consequence of the new findings contained in this thesis. Let us enumerate in a non-exhaustive manner those that we consider the main open research problems, most of which we plan to address in the near-term:

- **Complex evaluation process.** Currently, we carry out a very simple check that prevents us from falling into the most obvious inconsistencies, but apart from being insufficient, this process is embedded in the logic of the design tools that we have created (i.e. it is not explicit at all). One of the key extension points for the proposed description framework should be the definition of an explicit set of rules that allow us to perform an evaluation process for of the designs and annotations made with the framework.

- **Resources top-level ontology.** A top-level ontology of resources should be developed and it should describe different types of resources, the relationships among this classes of resources (e.g. aggregation, derivation, etc.), its nature, such as transient, unchangeable, volatile, etc.; its format (if it applies), such as XML, plain text expressed in ISO 8859-1, use policies, etc.; and any other facet that describes precisely the resource. The purpose for the extension of resources description is to the additional knowledge that would allow more advanced and dynamic handling of resources.

Regarding the architecture, we consider the work that we have carried so far in Vega as the blueprint of a much broader work that we will perform in the near future. The most important additions that we plan to introduce are:
Conclusions and Future Lines

- **Adaptable topology.** This is the main restriction of Vega so far, and such should be amended. As it is now, the topology of the system is set in advance. The system flexibility resides on detecting whether brokers are overloaded and adapting the level of detail of their notification activity. The next step is to make flexible the number of brokers and the connexion of these brokers. This activity involves the spawning of new brokers and the establishment of new links among them.

- **Spawning of new routers.** The first topology change that we plan to include is the spawning of brokers, especially inner brokers. Once a broker becomes overwhelmed due the size of its routing knowledge base, the system could spawn a new brokers following different criteria in order to receive part of the original broker routing knowledge base.

- **Discovery of new brokers.** The inclusion of advertisement expertise in the architecture was initially made to enable this activity. The rationale behind new brokers discovery is that when an inner broker is in an underused state, it should proactively discover new brokers of its interest (i.e. find brokers which advertisement expertise match to a given extend its subscription expertise). With this knowledge-guided criterion, new links might be created and the knowledge overlay network consequently updated.

- **Distributed event detection for an improved reactivity.** In the presented work we have focused on introducing a fine-grained distribution of the brokering activity, since our main target was scalability. The next logical step seems to focus in improving the system reactivity. As we have already exposed in the chapter devoted to the architecture, event-driven architectures reactivity is determined by its event detection architecture. We plan to move to a distributed event detection architecture, where complex eventflow patterns are divided and disseminated forming a distributed event detectors topology. This distributed detection system would foster the reutilization of intermediate detection results, and would reduce the detection time. And a reduction in detection time is a synonym of system reactivity improvement.

  In such approach, the eventflow patterns remain identical (the advantage of working at knowledge level shows up again). Furthermore, the core of the formalization of these patterns remains almost identical; the set of generated vocabularies and rules is almost unaltered. The key difference is that we should move from a single-agent abstract state machine representation to a multi-agent one. That implies that the execution of the detection machine involves shared locations use, but more importantly, the execution of several simultaneous abstract state machines might update those locations asynchronously.
Therefore we have to provide a solution that avoids inconsistencies in shared locations, without having to share too much information among distributed detector. This issue of event detection consistency, along with how to partition and distribute efficiently each subpart of the complex eventflow patterns, is part of our future research work.
Chapter 8  Appendixes

8.1  Appendix I: Knowledge Representation Model Elements

8.1.1 Ontology

According to Gruber (Gruber, 1993) and Borst (Borst, 1997) an ontology is a formal, explicit specification of a shared conceptualisation. As explained by Studer and colleagues (Studer et al., 1998) conceptualisation refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group.

Figure 8-1 displays a simplified version of knowledge model being use by WebODE. The Ontology concept is related with different elements that conform the conceptualization defined by the ontology. That includes zero to many instances of the concept Concept (related by the term relation hasConcept); zero to many TermRelation (using the hasTermRelation relationship); from zero to several instances of the concept Expression that represent the axioms that formalize the ontology (related with the Ontology concept using the hasAxiom term relation)
An ontology in the proposed framework is composed by the following building blocks:

- **Concepts.** The nature of a concept covers a wide spectrum of entities. Concepts may represent abstract entities (intentions, beliefs, feelings, etc.), specific concepts (people, computers, tables, etc.). They may also represent atomic or complex entities; real or fictitious. Moreover, they can describe static entities or dynamic ones, such as tasks, functions, processes, etc. We can also characterize concepts as sets, such as the case of Description Logics (see (Baader et al., 2007)). Concepts become thus groups of individuals that share some properties. More pragmatically, in the Firenze Framework concepts constitute the basic building elements of the terminologies of both the general-purpose ontologies and models; and for the problem-specific and domain-dependant ontologies. Concepts are further characterized by defining their attributes, their formal definition and meta-knowledge term.

- **Attributes.** Attributes refine concepts, allowing us to make further statements about the properties of a concept. Each attribute has a set of properties, in a similar fashion to the facets that describe slots in classical frame-based representations (such as those described in (Chaudhri et al., 1998)). For each attribute we define its type; its cardinality constraints, the minimum and maximum number of values that can be assigned to the attribute; the default value of attribute; the unit of measurement and its precision, in case that it is a numerical type that represents some physical phenomenon; the minimum and maximum values, in the case that it is a totally ordered type or an enumeration of symbolic values; and default value, in case that it is applicable. We distinguish between two types of attributes, namely class attributes and instance attributes.
  - **Class attributes.** Class attributes are attributes that define a concept itself. All the instances that belong to this class have the same value for each of its instances.
  - **Instance attributes.** Instance attributes are those that are defined in the concept but that take values in its instances (Fernández-López et al., 1999). Therefore each instance may define different values for this kind of attributes, in contrast with class attributes.

- **Formalization.** The formalization of a concept consists of a group of formulas that comprise the axiomatization of the definition of a concept.

- **Expressions.** In order to formalize facts in the ontology (as we have already expressed), and in order to express conditions that we will describe later on, such as precondition, conditions, assumptions, effects, etc. it is necessary to define axioms and formulas.
Nevertheless such expressions depend on the particular logical language chosen to define expressions (as depicted in Figure 4-70) and therefore are dependant of concrete use of the framework.

- **Constants.** A constant represents a quantity assumed to have a fixed value in a specific context. They represent constant values of any type. This value can be given an atomic value, or can it can be given by means of an expression that only involves other constants. Apart from its value, we should also define its data type; the unit of measurement and its precision in case that it is a constant that represents some physical phenomenon;

- **Term relations.** Term relations represent a type of *ad hoc* association between concepts of the domain and they are used to model interdependencies between several concepts. We allow the representation n-ary relations, nevertheless the involved concept should belong either as part of the domain or the range of the relation, and should be tagged with a unique label to be addressed when formalizing the term relation. For each term relation we define its signature, the type of each concept in its range/domain; its cardinality constraints, which are two integers that represent the minimum and maximum number of instances of the relation that should be made among individuals of the signature; the term relation properties, which can be either (e.g. symmetric, antisymmetric, reflexive, etc.) or domain specific; and finally (and optionally), its formalization, a group of formulas that comprise the axiomatization of aspects of the term relation.

- **Taxonomic relations.** Taxonomic relations are a subtype of term relations, a predefined set used to built concept taxonomies and hierarchies. Each of these relations posses precise semantics defined *a priori* in a domain independent manner. We consider the following taxonomic relations:

  - **Subclass-Of relation.** A concept is a subclass of other concept (referred as superclass) if each of its instances are necessarily instance of the superclass. As shown in Figure 8-3.a it is represented as an arrow traced from the subclass (\(\text{concept}_b\)) to the superclass (\(\text{concept}_a\) in the example).

Figure 8-2 Term relation graphical representation.
Appendixes

• **Disjoint-Decomposition.** A Disjoint-Decomposition of a concept (e.g. \textit{concept}_a, in Figure 8-3.b) is a group of subclasses of that concept that do not have common instances; but that do not cover entirely this concept. That means that there can be instances of the parent concept that are not instances of none of the decomposition.

![Disjoint-Decomposition](image)

Figure 8-3 Taxonomic relations among concepts.

- **Exhaustive-Decomposition.** An Exhaustive-Decomposition of a concept (e.g. see \textit{concept}_a, in Figure 8-3.c) is a set of subclasses of this concept that cover \textit{concept}_a and may have common instances and subclasses. In other words there cannot be instances of the concept that are not also instances of at least one of one of the concepts in the decomposition (instances of at least one of the concept that belong to the group of concepts \textit{concept}_i \ldots \textit{concept}_j).

- **Partition.** A Partition of a concept is a set of subclasses of that concept that does not share common instances; and moreover this group of subclasses exhaustively covers the parent concept. This means that there are not instances of the parent concept that are not instances of one (and only one as they are disjoint) of the concepts in the partition. Figure 8-3.d depicts how we represent a partition.

![Partition](image)

Figure 8-4 Is-A Graphical representation.
o **Is-A.** In contrast with the other taxonomic relationships, the Is-A relationship is established between a concept and an individual that we refer as instance. As shown in Figure 8-4 it is represented as a dashed line.

o **Groups.** In certain occasions we are interested in defining a relationship between a knowledge element and a group of knowledge elements. For example, in the previously described Taxonomic Relations we associated a concept with a group of concepts (see for example Figure 8-3.b to Figure 8-3.d).

o **Properties.** Relations defined in an ontology may have different properties. They can be the classical mathematic properties, namely reflexivity, symmetry, transitivity, etc.; or other *ad hoc* properties that the knowledge expert wants to define in order to capture a domain specific property. In the later case, this properties are associated with an Expression instance that formalizes the property.

Finally, we associate each of the ontology elements with a meta-knowledge term in order to identify and characterize them uniquely (see section 8.1.3 Meta-Knowledge Term).

8.1.2 Instances Set

An instance set holds all the descriptions of the individuals defined as instances of the concepts on the ontology; and the concrete instantiations of the term relations. An ontology specifies the structure of knowledge (entity types and relationships) and classification scheme. Instances set complements that knowledge, with the information of concrete individuals, constituting a knowledge base.\(^{38}\) Instances set store two different types of instances:

- **Concept instances.** Concept instances describe individuals that we have identified as instances of a concept of the ontology. Each instance is identified and described by means of a meta-knowledge term; the type of the instance, which is specified using the Is-A relationship; and its attribute values, a multiset of pairs \((\text{instance attribute identifier, value})\) that represent the values of these instance attributes for that individual.

- **Relations Instances.** Relation Instances are similar to concept instances. Each instance is identified and described by means of a meta-knowledge term; the type of the instance, which is specified using the Is-A relation; the origin of the relation instance, which is a set

\(^{38}\) You can also think of it as a Description Logic theory. It is divided into two parts: the TBox and the ABox. The TBox contains intensional (terminological) knowledge in the form of a terminology and is built through declarations that describe general properties of concepts. The ABox contains extensional (assertional) knowledge, which is specific to the individuals of the discourse domain (Baader et al., 2007).
of concept instances URIs; and the destination, which is the set of URIs of the range instances.

8.1.3 Meta-Knowledge Term

Meta-knowledge terms are used to state identification, description and additional authoring information about a knowledge element. We adapt the Dublin Core annotation vocabulary (Weibel et al., 1998), using the elements that do not specify or set any constraint on the specification language that (such as language, etc.). We consider the attributes and relations:

- **Content identification and description information.** It includes the attribute identifier that correspond to the URI where that uniquely identifies the element; title, a name given to the resource in a human readable format; description, a textual description in natural language of the knowledge element; subject, which represents the topic of the resource, represented using keywords, key phrases, or classification codes from an controlled vocabulary or taxonomy (such as ACM topics hierarchy\(^{39}\)); version, which assigns unique version names or unique version numbers to unique states of a knowledge element; and coverage, the extent or scope of the content of the element (e.g. some temporal interval, geographical situation, etc.).

- **Content creation and authoring information.** This group of attributes and relations depict information about the process of creation of the knowledge element and the role of the different entities involved in that conception process. We basically define three role relationships, the *hasCreator*, which relates the element with the entity which is the primarily responsible of making the knowledge element; *hasContributor*, which relates the elements with the different parties that may somehow contributed in the creation of the element; and the *hasPublisher* relation that links the knowledge element wit the entity responsible for provisioning and making the knowledge element available. We also define relations to express the knowledge element provenance relationships of the knowledge element, the *hasSource*, which associates the knowledge element with the set of resources from which the element is directly derived; and *hasRelationWith*, which relates the knowledge element with resources that are related with it.

---

8.2 APPENDIX II: BINARY DECISION DIAGRAMS

A binary decision diagram is a data structure that is used to represent a Boolean function. Moreover, as described in (Bryant, 1986) they can easily in a systematic manner transformed in to canonical representation of these functions. Binary decision diagrams were firstly introduced by (Lee, 1959)(Boute, 1976)(Akers, 1978) and they have been well studied and applied to a great variety of computer science problems.

In this section we summarize the basic binary decision diagrams theory in order to be able to understand the architecture chapter. We take a functional programming approach both in the description of the tree structures in the pseudo code of the algorithms with the idea of making these descriptions more intuitive.

Binary decision diagrams are rooted, directed acyclic graph composed by two sets \( \langle N, A \rangle \) where:

- **The set \( N \) of nodes.** A set of variable nodes (we usually refer to them simply as nodes because since using the term variable may result misleading) with an output degree of two, but with an arbitrary input degree. We define for each node in a binary decision diagram:
  
  - **Atomic formula nodes.** We consider each node as an atomic formula, represented as a triplet of the form \( \text{Node} = \langle \text{Variable, Operator, Value} \rangle \) (with its respective accessor functions, such as \( \text{variable : Node} \rightarrow \text{Variable} \), \( \text{operator : Node} \rightarrow \text{Operator} \), \( \text{value : Node} \rightarrow \text{Value} \)).

  - **Terminal nodes.** One two terminal nodes labelled 0 or 1. Obviously, 0 corresponds to false and 1 corresponds to true.

- **The set \( A \) of arcs.** The set of arcs is a relation over nodes (i.e. \( A = \{ \langle \text{Node, Node} \rangle \} \)) is determined by two functions \( \text{low(node)} \) (i.e. \( \text{low : Node} \rightarrow \text{Node} \)) and \( \text{high(node)} \) (i.e. \( \text{high : Node} \rightarrow \text{Node} \)). This functions return in a given node which will be the next node depending on whether the atomic formula holds (the high function) or not (the low function).
Apart from the binary decision diagram structure and functions, we define two additional functions that allow us to evaluate if for a given variable assignment the Boolean function represented by the tree holds. They are the node and binary decision diagram evaluation functions.

- **Node evaluation function.** A node evaluation function $\alpha$ determines whether the expression represented in a given node holds under a concrete variable assignment. A variable assignment can be considered a set of pairs of the form $\text{VariableAssignation} = \{(\text{Variable}, \text{Value})\}$. The node evaluation function, that we will note it as $\alpha_{\text{assignation}}$ with signature $\alpha : \text{Node} \times \text{VariableAssignation} \rightarrow \{\text{true}, \text{false}\}$, simply bounds the variable of the node with the value present in the variable assignment, and checks whether the expression holds.

- **Binary decision diagram evaluation function.** The basic algorithm to evaluate whether a given logical expression represented as a binary decision diagram holds is the following:

\[
evalBDD(bdd, \alpha_{\text{assignation}}) \equiv evalNode(root(bdd), \alpha_{\text{assignation}})
\]

\[
evalNode(0, \alpha_{\text{assignation}}) = \text{false}
\]

---

40 In some articles it is referred as an assignation function, since assigns Boolean values to the diagram nodes under a given variable assignment. We believe that the verb evaluate captures better what this function does.

41 Independently of the extensions to the basic version of binary decision diagrams that we will describe in this introduction, the evaluation algorithm remains unaltered.

42 The classical and efficient version of this algorithm is depicted using an specific array-based data structure. We prefer to provide this functional description, as we believe it portraits better the algorithm.
evalNode(1, αassignation) ≡ true

\[
\text{evalNode}(\text{node}, \alpha_{\text{assignation}}) \equiv \left[ \alpha_{\text{assignation}}(\text{node}) \land \text{evalBDD}(\text{high}(\text{node}), \alpha_{\text{assignation}}) \right] \\
\lor \left[ \neg \alpha_{\text{assignation}}(\text{node}) \land \text{evalBDD}(\text{high}(\text{node}), \alpha_{\text{assignation}}) \right]
\]

Binary decision diagrams are much more useful when we impose some restrictions to its structure and nodes. A binary decision diagram is referred as ordered (Bryant, 1986) if on all paths through the graph the variables respect a given linear order \text{node}_1 < \text{node}_2 < \ldots < \text{node}_N. An ordered binary decision diagram is reduced if the following properties hold:

- **Uniqueness.** Not two distinct nodes \( u \) and \( v \) have the same variable name and low and high successors

  \[
  \text{var}(u) = \text{var}(v) \land \text{low}(u) = \text{low}(v) \land \text{high}(u) = \text{high}(v) \rightarrow u = v
  \]

- **Non-redundant** no variable node \( u \) has identical low and high successor

  \[
  \forall \text{node} \in \text{Nodes}(\text{BDD}) \rightarrow \text{high}(\text{node}) \neq \text{low}(\text{node})
  \]

Reduced ordered binary decision diagrams (known as ROBDD in the literature) have several properties that make them a good choice for representing logic expressions, such as the fact that they are canonical (i.e. a one to one relationship between a logical expression and its representation as a binary decision diagram); they are much compact and efficient to handle than ordinary ones, etc.

### 8.2.1 Binary decision diagrams in publish/subscribe systems

The use of Binary Decision Diagrams in publish/subscribe systems was introduced in (Campailla et al., 2001) for the efficient filtering of notifications. The key concept in the approach of Campailla and colleagues is to consider a notification of an event as a variable assignation for binary decision diagrams. Binary decision diagrams with this approach become the representation of a subscription with the following signature (i.e. the same that we have already show, but it includes the possibility of leaving a variable undefined) \( \alpha : \text{Node} \times \text{VariableAssignation} \rightarrow \{\text{true}, \text{false}, \text{undefined}\} \) to the nodes of the binary decision diagram.
8.3 **Appendix III: Abstract State Machines**

In this appendix we introduce basic information about abstract state machines that we find helpful to understand section 5.5 The Event Detection Architecture. We take a pragmatic approach to define them, as we want to make it easy to understand. Nevertheless we remit the reader to the work of Gurevich and colleagues, specially the papers where they demonstrate that any algorithm (both sequential (Gurevich, 2000) and parallel (Blass and Gurevich, 2003)) can be expressed using an abstract state machine.

Abstract state machines (originally called evolving algebras (introduced in (Gurevich, 1988), and thoroughly described in (Gurevich, 1995))) can be thought of as a state machine which states are defined as first-order logic structures. And moreover, the transition between these states are specified by the updates to these first-order logic structures, specified by mean of a transition rules set.

As described in (Börger and Stärk, 2003) (and as we represent in Figure 8-6), abstract state machines are divided into two main categories depending on its nature, namely single-agent (or basic) abstract state machines and multi-agent abstract state machines. Multi-agent abstract state machines allow formalizing the actuation of the system in terms of multiple entities that are collaborating to achieve the machine functionality. The latter can be further divided in two categories: synchronous and asynchronous multi-agent abstract state machines, depending if they assume an implicit synchronization among the system, or they opt for a more realistic approach where collaborating entities act at will without explicit synchronization.
In this section we only describe single-agent abstract state machines (also known as basic abstract state machines) as these are the ones used in the chapter. First of all we need to describe in more detail the concept of abstract state (section 8.3.1 Abstract States). After that, we introduce the concept of transition rules and the simple catalogue of transition rules that we have used in this chapter (8.3.2 Transition Rules). Finally, we describe how an abstract state machine defined using these elements is actually executed (8.3.3 Abstract State Machine).

### 8.3.1 Abstract States

As we have outlined, states of an abstract state machine are defined as a mathematical logic structures. A signature, the base set of the state, and its interpretation, therefore basically composes a state. More precisely:

- **The signature of the state.** The signature (or vocabulary and usually noted Σ) of an abstract state machine is a finite collection of fixed arity function names (also referred as locations in the literature). Each function name f has an arity greater or equal to zero, which determines the number of arguments that the function can take. Nullary function names are called constants. Function can be static or dynamic.

- **Static functions.** They correspond the common notion of function in mathematical logic. They never change during a run of a machine; given a term as input they always yield the same result.

- **Dynamic functions.** These functions are classified as in (or monitored); controlled; shared (or interaction); or out. There are functions neither updatable by the machine or the environment but which are defined in terms of other static and dynamic (and derived) functions.
- **Controlled functions.** These functions are only updatable by the rules of the abstract state machine (we will describe how in section 8.3.2 Transition Rules). That means that they can neither be read nor updated by any external entity of the environment.

- **In (or monitored) functions.** This type of functions represents those that can only be updated by the environment. Nonetheless, they can be read by the abstract state machine, becoming the nexus of the machine with its environment.

- **Shared functions.** These functions can be read and also updated by both the environment and the rules of the abstract state machine. They are used to represent shared variables among different machines.

- **Out functions.** They can be updated but not read by the abstract state machine. Nevertheless, they can be read by the external entities that belong to the environment, as it can be easily inferred they become the complement to in function.

All functions have associated locations\(^{43}\). Suppose a function \(f\) with arity \(N\), and a set of ground terms \(t_1, \ldots, t_N\). The location for this function of a state \(S\) is defined as the pair \((f, a_1, \ldots, a_N)\), where \(a_1, \ldots, a_N\) are the interpretation of each of the terms \(t_1, \ldots, t_N\). The content of that location is what the function returns when the function is applied to those terms.

\(^{43}\) Going back to Figure 8-7, in this picture we referred both to functions and locations, as they are so intimately related.
It can be referred as $t^S(t_1,\ldots,t_N)$, and abbreviated as $S(t)$. You can think of the tuple $(t,a_1,\ldots,a_N)$ as the address to the location that holds the function value for $a_1,\ldots,a_N$.

Finally, we define a last type of function, relation functions. These functions range is Boolean, (i.e. the values stored in the function location are either true or false); they represent the same entities as predicates did in first-order logic structures.

Finally, the terms of the signature are defined in the same manner as usual, variables are terms; constants that belong to the signature are terms; and if the name $f$ belongs to the signature as an function with arity $N$, and $t_1,\ldots,t_N$ are terms, then $f(t_1,\ldots,t_N)$ is a term. Terms that contain no variable are referred as ground terms.

- **The base set of the state.** The base set of a state (also referred as superuniverse, noted as $|S|$), contains the elements of a state (i.e. the elements of the domain of the abstract state machine). The only restriction is that they must always contain elements true, false, undef and at least one element for each of constants interpretation. Apart from these, $|S|$ can contain numbers, strings, etc., it is up to the application domain.

- **The interpretation of the state.** It is defined in the same manner as a first-order logic interpretation, and thus an interpretation is defined as a function. But before defining the interpretation function of an abstract state $S$, we have to define variable assignments. Variable assignment for $S$ is a finite function which assigns elements of $|S|$ to a finite number of variables. $[x \mapsto a]$ denotes a variable assignment that assigns the element $a$ to the variable $x$ (and leaves the other variables unaltered). In a formal fashion it is written as follows:

$$
\sigma[x \mapsto a](y) = \begin{cases} 
  a & y = x \\
  \sigma(y) & \text{otherwise}
\end{cases}
$$

Depending on the nature of the term that is applied the interpretation function its interpretation is determined as follows:

- The interpretation of a variable is the value that we have already assigned to it.

$$
[[x]]^S_S = \sigma(x)
$$

- The interpretation of a constant is its invariable value in that specific state.

$$
[[c]]^S_S = c^S
$$
The interpretation of a function with a set of term arguments \( t_1, \ldots, t_N \), is the interpretation of the application of that function to the interpretation of each of these terms.

\[
\sigma^S \left[ \left[ f(t_1, \ldots, t_N) \right]^S \right] = \left[ f \left( \left[ t_1 \right]^S, \ldots, \left[ t_N \right]^S \right) \right]^S
\]

### 8.3.2 Transition Rules

Let us now depict how an abstract state machine transits between its different states. Transitions are defined by means of transition rules. The most basic rule is a function update.

\[
f(t_1, \ldots, t_N) := t
\]

Being \( f \) a function name of arity \( N \), and \( t_1, \ldots, t_N \) ground terms. This assertion might look quite odd at first sight, since functions are meant to return terms. But what that expression really means for an abstract state machine is that the location \( f(t_1, \ldots, t_N) \) is updated with the value of the term \( t \) (i.e. \( \left[ f(t) \right]^S \)). Location updates are specified by the location and the value to be stored, as the pair \( (l, v) \). The collection of updates that certain transition rule might involve receives the name of update set. To transit between two states means to fire that update set simultaneously. The value for each of the locations of the resulting state is expressed as follows (we use the same notation that appears in (Börger and Stärk, 2003):

\[
(S+U)(l) = \begin{cases} 
    v & \text{if } (l, v) \in U \\
    S(l) & \text{otherwise}
\end{cases}
\]

Graphically we represent the update of a state as an automaton that can be seen as:

![Figure 8-8 An abstract state machine as an automata.](image)

Once we have introduced the simplest transition rule, let us enumerate those that we had used to define the abstract state machines. They are just combinations of the simplest ones, since at the end transitions are just set of locations updates. Therefore, the transition rule has the following form:

\[
RULE \equiv f(t_1, \ldots, t_N) := t
\]

The first one is just the addition of a condition guard to the already described update.
The first one is just the addition of a condition guard to the already described update.

The other transition rule that we use is the for all rule

8.3.3 Abstract State Machine Execution

Finally, once we have defined all the elements, we describe how the execution of an abstract state machine is performed, as this is how they are used in Vega to detect composite events. The description we provide is based on the one that appears in (WSMO D14, 2006), which we have found very intuitive and pedagogic. A single abstract state machine execution step can be summarized as follows:

- **Unfold the rules.** Evaluate the guards/conditions according to the current state. This step is repeated until no condition of any rule evaluates to true. In case that the unfolding yields an empty update set, the execution is halted.

- **Execute simultaneously all the updates.** The set of updates has been selected in the previous steps are executed, which means the following:
  - **Update locations.** If the updates are consistent (i.e. no two different updates update the same location with different values, i.e. it does not exist two updates for location loc such as update(loc,v), update(loc,v') with v ≠ v'). The result of the enactment of all these
Appendixes

updates yields the next state. Note that the abstract state machines that we have created (see 5.5.2.1 Complex Events Detection) have been carefully formed so that they do not lead to an inconsistent update (at the cost of introducing some inessential locations). In case of any inconsistency, the execution should be halted immediately yielding an error.

- **All locations that are not affected by updates, keep their values.** As we mentioned when we enumerated the reason that lead us to choose abstract state machines, they focus just on updates, overcoming thus the frame problem.

8.4 **Appendix III: SWAP, A Distributed Peer-to-Peer Knowledge Base**

The SWAP project’s44 objective was to combine the semantic Web and peer-to-peer computing; developing the technology that is necessary to allow users their individual views on knowledge, and let them share knowledge effectively. In the context of this project the SWAP architecture was proposed. The SWAP architecture (Haase et al., 2004)(Haase et al., 2008) aims at decentralizing knowledge bases, creating a peer-to-peer network of distributed knowledge bases and an intelligent algorithm to propagate queries.

The key concept in the SWAP architecture is expertise. Expertise is defined as the abstract semantic description of the knowledge base of a peer in of terms from the common ontology. The common ontology describes the domain and is shared by all the peers. Peers share their expertise with other peers forming what is referred as the semantic topology, a directed graph where each node corresponds with a peer, and a set of arcs between them represent awareness relationship of peers (i.e. if an arc has been established from a peer to other, it means that the first peer is aware of the expertise of the destination peer). The semantic topology is formed dynamically in a decentralized fashion; each of its peers may decide to add new peers to their adjacent neighbourhood. In order to discover such candidates, peers are continuously listening to advertisement messages that describe the capabilities of other peers in the form of (Peer, Expertise) tuples. When a peer receives one of these advertisements it can freely decide whether to add this peer to its neighbourhood or simply ignore them.

This semantic topology combined with an expertise-based peer selection algorithm form the basis for intelligent query routing. The rationale of the expertise-based routing of queries is to

---

send queries only to those peers that are more likely able to answer them; in terms of expertise, is to send queries only to those which expertise matches with the concepts described in the query. More precisely

- When a peer receives a query it first tries to find relevant results on its own knowledge base. After that, each peer must decide to which of the nodes its neighbourhood it delivers the query in order to recollect more results that are later combined with those obtained locally.

Figure 8-9 Expertise Based Matching.

- In order to determine the most suitable candidate peers that should receive the query, an abstraction and similarity matching processes are performed, as shown in Figure 8-9, in order to find. More precisely:

  • **Abstraction process.** The first step is to abstract the subject from the query. The subject is an abstraction of a given query expressed in a set of terms of the ontology share by all the peers. As described in (Haase et al., 2008), the subject can be seen as a complement to an expertise description, as it specifies the required expertise to answer the query.

  • **Similarity matching process.** Once the subject of the query has been abstracted, the subject is compared with the expertise of each of the peers on its neighbourhood; for each candidate a similarity matching is performed (as shown in Figure 8-9). This comparison is made using a subject/expertise similarity function $\text{SimilarityFunction} : \text{Subject} \times \text{Expertise} \rightarrow [0,1]$ that yields the semantic similarity between the subject of a query and the expertise of a peer. The value 0 represents no similarity at all, and 1 means that the peer is an ideal candidate for answering that query. Using the results of the similarity function, peers a ranked an ordered. Depending on the peer delivery policy, the query is sent to those which similarity is greater that certain threshold, to the peer which similarity is higher, to a given number of most similar peers, etc.
Appendixes

SWAP does not provide any specific implementation or representation of the above-presented ideas, it just identifies the main elements, relationships and processes necessary for the intelligent query routing in an abstract fashion. Bibster (Haase et al., 2005), a peer-to-peer system for exchanging bibliographic data among researchers, is the instance of SWAP, and in consequence provides an implementation of all these elements. It defines every concept that key characteristics with respect to SWAP are:

- **Common ontology.** Bibster common ontology was either the Semantic Web Research Community Ontology (SWRC)\(^45\), which models among others a research community, its researchers, research topics, publications, tools, etc.; or the ACM Topic Hierarchy that describes specific categories of literature for the Computer Science domain.

- **Expertise.** The expertise \(E\) of a peer is the set of concepts for which a peer provides classified bibliographic references instances (i.e. \(E \subseteq 2^C\), where \(C\) denotes a set of concepts that belong to the common ontology).

- **Subject.** The subject \(S\) of a query is defined as the set of concepts that are referred in the query (i.e. \(S \subseteq 2^C\), where \(C\) denotes a set of concepts that belong to the common ontology).

- **Similarity function.** The similarity function defined in Bibster is the following

\[
\text{SimilarityFunction}(S, E) = \frac{1}{|S|} \sum_{C_i \in S} \max_{C_j \in E} \text{SimilarityFunctionConcepts}(C_i, C_j)
\]

Basically it is the promethium of the similarity of each of the concepts of the subject of the query and the most similar concept present in the peer expertise. The \(\text{SimilarityFunctionConcepts}\) used is the similarity function between concepts defined in (Li et al., 2003) for measuring the similarity between concepts in a hierarchical structured semantic network:

\[
\text{SimilarityFunctionConcepts}(C_1, C_2) = \begin{cases} e^{-\alpha l} & \text{if } c_1 = c_2 \\ e^{-\beta h} & \text{if } c_1 \neq c_2 \end{cases}
\]

Where \(l\) is the length of the shortest path between concepts \(C_1\) and \(C_2\), \(h\) is the level in the tree of the direct common subsumer from \(C_1\) and \(C_2\), and \(\alpha, \beta \geq 0\) are parameters for weighting the contribution of \(h\) and \(l\) respectively (after running several experiments while implementing Bibster, they concluded that \(\alpha=0.2\) and \(\beta=0.6\) where the optimal values).

\(^{45}\)http://ontoware.org/swrc/
Chapter 9  Bibliography

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Benjamins V.R., Pierret-Golbreich C., (1996), Assumptions of Problem-Solving Methods, Advances in Knowledge Acquisition</td>
<td></td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>1997</td>
<td>Borst W. N.</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Title</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Box et al., 2000</td>
<td>Simple Object Access Protocol (SOAP) 1.1 W3C Note. Available at <a href="http://www.w3.org/TR/SOAP">http://www.w3.org/TR/SOAP</a></td>
</tr>
<tr>
<td>Brickley and Miller, 2007</td>
<td>FOAF Vocabulary Specification 0.91 Available at: <a href="http://xmlns.com/foaf/spec/">http://xmlns.com/foaf/spec/</a> RDF Available at: <a href="http://xmlns.com/foaf/spec/index.rdf">http://xmlns.com/foaf/spec/index.rdf</a></td>
</tr>
<tr>
<td>Bryant, 1986</td>
<td>Graph-Based Algorithms for Boolean Function Manipulation</td>
</tr>
<tr>
<td>Carzaniga et al., 1998</td>
<td>Issues in supporting event-based architectural styles.</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td></td>
</tr>
<tr>
<td>Empty</td>
<td></td>
</tr>
<tr>
<td>(Christensen et al., 2001)</td>
<td>Christensen E., Curbera F., Meredith G., Weerawarana S. (2001) <em>Web Services Description Language (WSDL) 1.1 W3C Note</em>, available at <a href="http://www.w3.org/TR/wsd1">http://www.w3.org/TR/wsd1</a></td>
</tr>
<tr>
<td>(Clancey, 1985)</td>
<td>Clancey W. J. (1985) <em>Heuristic Classification</em>. Artificial Intelligence,</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Reference</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>Davies J., Domingue J., Pedrinaci C., Fensel D., Gonzalez-Cabero R., Potter M., Richardson M., Stincic S.</td>
<td>(Davies et al., 2009) Towards the Open Service Web, BT Technology Journal, Volume 26 (2) Springer,</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Title</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Farquhar et al., 1997</td>
<td>The Ontolingua Server: A Tool for Collaborative Ontology Construction</td>
</tr>
<tr>
<td>Reference</td>
<td>Location</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Reference</td>
<td>Citation</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Year</td>
<td>Author(s)</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2007</td>
<td>Springer Book Series, Knowledge and Data Management in Grids</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Reference</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>(OWL, 2004)</td>
<td>OWL Web Ontology Language Overview W3C Recommendation 10 February 2004 <a href="http://www.w3.org/TR/owl-features/">http://www.w3.org/TR/owl-features/</a></td>
</tr>
<tr>
<td>Reference</td>
<td>Details</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>(WSDL 1.1, 2001)</td>
<td><em>Web Services Description Language (WSDL)</em> 1.1, W3C Note 15 March 2001 available at: <a href="http://www.w3.org/TR/wsd1">http://www.w3.org/TR/wsd1</a></td>
</tr>
<tr>
<td>(WSMO D14, 2006)</td>
<td>D14v0.2. <em>Ontology-based Choreography and Orchestration of WSMO Services</em>, available at: <a href="http://www.wsmo.org/TR/d14/v0.2/">http://www.wsmo.org/TR/d14/v0.2/</a></td>
</tr>
</tbody>
</table>
| (WSMO D29, 2005) | D29v0.1 *WSMO Mediators* WSMO Final Draft 21 December 2005,
<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>