Flexible Service Composition

Adam Barker

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2007

Abstract

Service-oriented architectures are a popular architectural paradigm for building software applications from a number of loosely coupled, distributed services. Through a set of procedural rules, workflow technologies define how groups of services coordinate with one another to achieve a shared task. A problem with workflow specifications is that often the patterns of interaction between the distributed services are too complicated to predict and analyse at design-time. In certain cases, the exact patterns of message exchange and the concrete services to call cannot be predicted in advance, due to factors such as fluctuating network load or the availability of services. It is a more realistic assumption to endow software components with the ability to make decisions about the nature and scope of their interactions at runtime.

Multiagent systems offer a complementary paradigm: building software applications from a number of self interested, autonomous agents. This thesis presents an investigation into fusing the agency and service-oriented architecture paradigms to facilitate flexible, workflow composition. This proposed agent-based approach to workflow composition is founded on the concept of shared interaction protocols that allow groups of agents to communicate in open systems. By adopting an agent-based approach to workflow composition, active autonomous agents can utilise the typically passive service-oriented architectures, found in Internet and Grid systems. In contrast with statically defined, centralised workflows, decentralised agents can perform service composition at runtime, allowing them to operate in scenarios where it is not possible to define the pattern of interaction in advance.

Application to real scenarios is a driving factor behind this research. By working closely with a number of active Grid projects, namely AstroGrid and the Large-Synoptic Survey Telescope (LSST), a concrete set of requirements for scientific workflow have been derived based on realistic science problems. This research has resulted in the MultiAgent Service Composition (MASC) language to express scientific workflow, methodology for system building and a software framework which performs agent-based web service composition, in order to enact distributed e-Science experiments. Evaluation of this thesis is conducted through case study, applying the language, methodology and software framework to solve a motivating set of workflow scenarios.

Acknowledgements

I would like to express my sincere gratitude for the expert supervision provided by Dr. Dave Robertson, Dr. Chris Walton, Dr. Bob Mann and Professor Austin Tate. Without their guidance, patience and advice this thesis would not have been possible.

During my time at the School of Informatics I have been integrated into a thriving research environment, I feel privileged to have been given the opportunity to work alongside such enthusiastic, passionate academics. Thanks to everyone at the Institute I have called home for the past three years; the Centre for Intelligent Systems and their Applications (CISA). Particular thanks to the men of office 4.15: Paolo Besana, Jarred Mcginnis, Guo Li, Fadzil Hassan and Tommy French, surely the finest office in all of Informatics.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Adam Barker*)

Publications List

Throughout the course of this degree, I have taken advantage of the immeasurably helpful input provided by the peer review system. Most of the ideas developed in this thesis have been presented or published for a number of conferences and workshops:

- Adam Barker and Robert G. Mann. Flexible Service Composition. In Lecture Notes in Artificial Intelligence, Volume 4149, pages 446-460, Springer Verlag, 2006.
- Adam Barker and Robert G. Mann. Agent-Based Scientific Workflow Composition. In Astronomical Data Analysis Software and Systems XV, pages 485-488, 2006.
- Adam Barker. Agent-Based Service Coordination For The Grid. In IEEE/WIC/ACM International Conference on Intelligent Agent Technology, pages 611-614, 2005.
- Adam Barker. Agents, Consumers of Service Oriented Architectures. In Proceedings of The First European Young Researchers Workshop on Service Oriented Computing, April 2005.
- C. Walton and Adam Barker. An Agent-Based e-Science Experiment Builder. In Proceedings of The 1st International Workshop on Semantic Intelligent Middleware for the Web and the Grid, European Conference on Artificial Intelligence (ECAI), Valencia, Spain, August 2004.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Building distributed systems is a difficult task; it has been claimed that such development projects are amongst the most complex construction tasks undertaken by humans [34]. With the adoption of pervasive network access and increased bandwidth there has been a trend towards building distributed systems using the service-oriented architecture (SOA) [45] paradigm. A service-oriented architecture is an information technology approach or strategy in which applications make use of (or rely on) services available in a network, such as the World Wide Web. A service provides a set of functionalities. This can be a single discrete function, such as converting between two currencies, or it can be composed from a set of inter-connected functions, such as the process of reserving a seat on a flight. Multiple services can be glued together to perform more complex operations, otherwise known as a workflow [31].

A problem with workflow specifications is that often the patterns of interaction between the distributed services are too complicated to predict and analyse at design-time [10]. In certain cases the exact patterns of message exchange and the concrete services to call cannot be predicted in advance, due to factors such as: changing network load or availability of software components etc. It is a more realistic assumption to endow software components with the ability to make decisions about the nature and scope of their interactions at runtime.

Multiagent systems offer a complementary paradigm for building complex distributed systems, and are currently the focus of much research. A multiagent system is composed of multiple interacting software entities, known as agents. Although the term agent has many competing definitions, it is generally accepted that an agent is a computer system that is situated in an environment, and is capable of autonomous action in this environment in order to meet its design objectives [73].

This thesis presents an investigation into fusing the agency and service-oriented architecture paradigms to facilitate flexible, workflow composition. More specifically the problem of service composition in a scientific domain (commonly known as scientific workflow) is addressed. This proposed agent-based approach to workflow composition is founded on the concept of shared interaction protocols that allow groups of agents to communicate in open systems. By adopting an agent-based approach to workflow composition, active autonomous agents can utilise the typically passive service-oriented architectures, found in Internet and Grid systems. In contrast with statically defined, centralised workflows, decentralised agents can perform service composition at runtime, allowing them to operate in scenarios where it is not possible to define the pattern of interaction in advance.

This thesis is a discussion of fusing the agency and service-oriented architecture paradigms. More specifically we addresses the problem of service composition in a scientific domain (commonly known as scientific workflow) and propose using a decentralised, agent-based approach to provide a flexible solution to the service composition problem. Active, autonomous agents can consume the typically passive service-oriented architectures, found in Internet and Grid systems, facilitating dynamic, runtime composition of services in scenarios where the patterns of interaction are too complex to define at design-time.

1.1 Contributions to Knowledge

An overview of the research presented by this thesis is illustrated by Figure 1.1, each of the individual contributions to knowledge will now be discussed in turn.

1.1.1 Requirements of Scientific Workflow

Scientific workflow has an extra set of requirements, which go beyond the functionality that traditional workflow languages and execution engines provide. There is a need to support the knowledge discovery and exploration processes which lead from a scientific hypothesis, to a concrete workflow specification. As as result of the push for

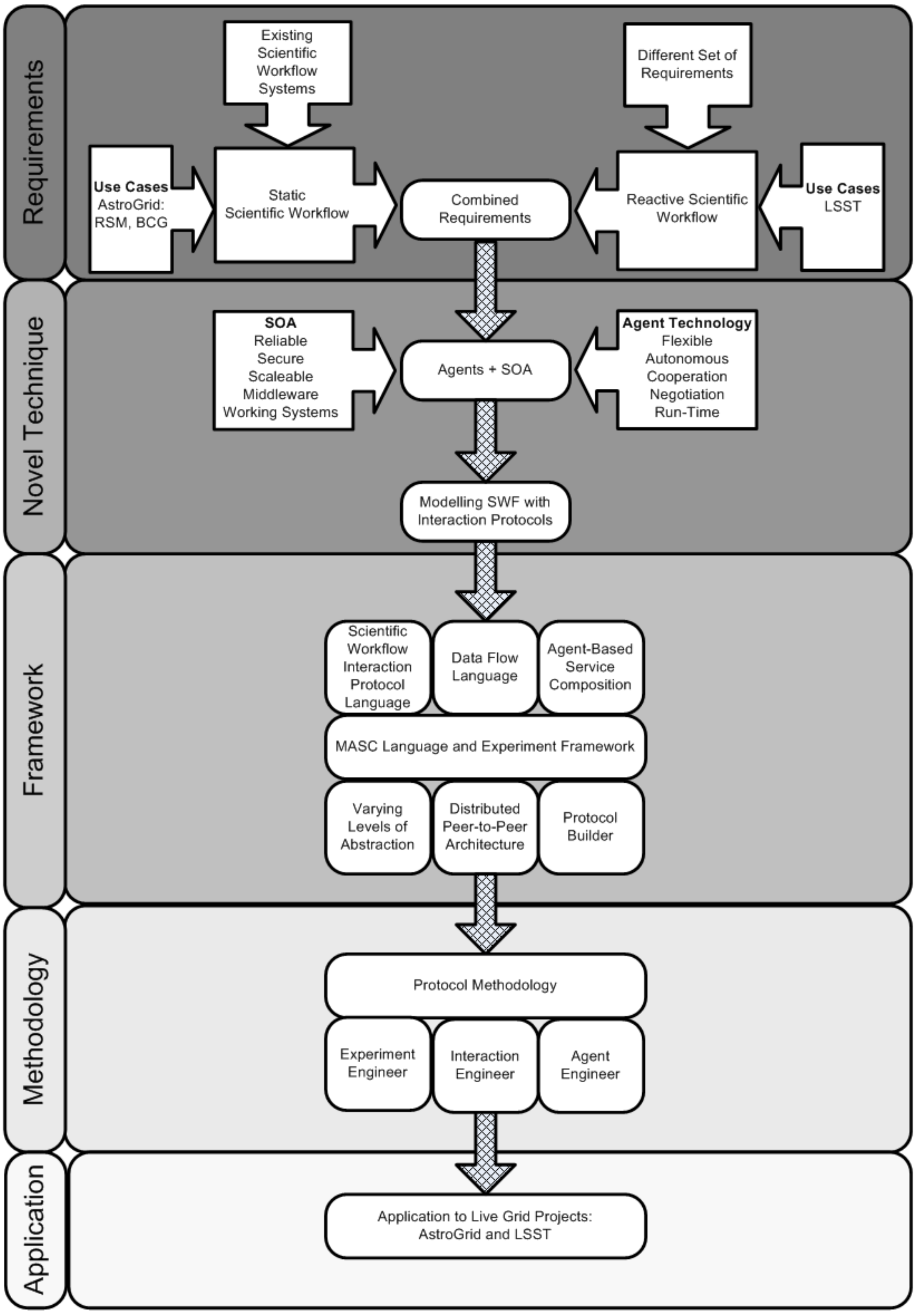

Requirements
Existing Scientific Workflow Systems
Different Set of Requirements
Use Cases AstroGrid: RSM, BCG
Static Scientific Workflow
Combined Requirements
Reactive Scientific Workflow
Use Cases LSST
Novel Technique
SOA Reliable Secure Scaleable Middleware Working Systems
Agents + SOA
Agent Technology Flexible Autonomous Cooperation Negotiation Run-Time
Modelling SWF with Interaction Protocols
Framework
Scientific Workflow Interaction Protocol Language
Data Flow Language
Agent-Based Service Composition
MASC Language and Experiment Framework
Varying Levels of Abstraction
Distributed Peer-to-Peer Architecture
Protocol Builder
Methodology
Protocol Methodology
Experiment Engineer
Interaction Engineer
Agent Engineer
Application
Application to Live Grid Projects: AstroGrid and LSST

Figure 1.1: Research overview

ubiquitous computing through e-Science and Grid technologies, there is an increased interest in this area of research. This is demonstrated by the currently running projects: myGrid [53], Imperial College e-Science Networked Infrastructure (ICENI) [41], Kepler [6] and Triana [54]. However, it is only recently that scientific workflow has become a sub-field of workflow, this research area is still relatively new and as a result there are very few languages targeted specifically at scientific workflow.

This thesis has worked closely with a number of active Grid projects, focused on Virtual Observatory engineering; namely AstroGrid [4] and the Large Synoptic Survey Telescope [56] (LSST). As a result of working with these projects a set of concrete workflow scenarios have evolved, based on: batch processing of pre-defined services, knowledge acquisition, knowledge discovery and runtime composition of services. These workflow scenarios act as a motivating factor behind this research work and are a valuable commodity in their own right. By understanding the processes behind these workflows and researching existing systems this thesis has been able to identify a concrete set of requirements for scientific workflow. This contribution is discussed in Chapters 2, 3 and 4 and illustrated by the first stage of Figure 1.1.

1.1.2 Service Composition through Interaction Protocols

The analysis of existing service composition techniques and workflow scenarios taken from the live Grid projects form the requirements analysis for this thesis. In order to meet these requirements this research views the service composition problem in a fundamentally different way. The flexible coordination technique of *interaction protocols*, from the field of multiagent communication, has been applied to the problem of *scientific workflow modelling*, found in the Grid community. This has allowed the typical features and requirements of a scientific workflow to be understood in terms of pure coordination and executed in an agent-based, decentralised, peer-to-peer architecture. Section 2 of Figure 1.1 illustrates this contribution.

1.1.3 Agent-Based Service Composition Language and Framework

The product of this research is a formal interaction protocol language and state-of-the-art web services composition framework to model and enact scientific workflows. The detailed breakdown of the language and framework will be discussed in Chapters 5 and 6, however to provide an overview (illustrated by section 3 of Figure 1.1), this framework offers:

- **Formal language:** The MultiAgent Service Composition (MASC) language is an agent-based solution to the service composition problem and is centred around the concept of interaction protocols. The language directly addresses the requirements of scientific workflow, discussed throughout this thesis.
- **Dataflow language:** Depending on the user, the MASC language can be utilised to model scientific workflow at varying levels of abstraction. Scientists don't want to concern themselves with the intricacies of protocol design, this is a time consuming and error prone task, due to the modelling of concurrent processes. To this end, a high level dataflow language has been designed which sits on top of the protocol layer. By treating protocol code as black boxes of computation, a scientist can compose an experiment from the top-down using the dataflow language to wire components together. An experienced engineer on the other hand can model experiments from the bottom-up, by writing protocol code which coordinates a group of agents and web services.
- **Agent-based service composition framework:** As well as providing a formal language, the `Zorro` framework is a full, state-of-the-art, open-source implementation of the concepts addressed by this thesis. Given a protocol and a group of web services, the execution engine allows protocol code to be executed dynamically, in a distributed, peer-to-peer environment.

1.1.4 Agent-Oriented Software Engineering Methodology

In addition to providing a language and framework for scientific workflow composition, this thesis also introduces the *coordination-oriented methodology* which provides users with guidance on how to build systems using these techniques (section 4 of Figure 1.1). This methodology is discussed in detail in Chapter 7 and categories system building at various levels of abstraction, a user can then adopt one of three distinct roles:

- **Experiment engineer:** A user at this level is concerned with the cycle of events for taking a scientific hypothesis and designing a workflow which attempts to prove or disprove that hypothesis. This is the most abstract layer, a user treats protocol components and services as parameterisable black boxes of computation.

- **Interaction engineer:** The primary concern of an interaction engineer is to take a software specification and divide it into a number of distinct agent roles, specifying the details of how these roles coordinate with one another (within a multiagent system) to achieve the overall aim of the specification.
- **Agent engineer:** This is the least abstract level and gives guidance on how engineers should construct individual, intelligent agents. This is achieved by integrating agent stubs with customised reasoning models, these reasoning models represent an agent's internal knowledge and can be invoked throughout the execution of an interaction protocol.

1.1.5 Application to Live Grid Project

Workflow scenarios have been a driving factor behind this thesis. Modelling these scenarios has allowed the language and framework to evolve and provided the project with a realistic application domain, as illustrated by section 5 of Figure 1.1. AstroGrid has served as a test bed, in order to verify and execute our ideas on a live framework, with live services and data. In Chapter 7 our language, framework and methodology are applied to each of the motivating workflow scenarios, demonstrating our agent-based approach to service composition.

1.2 Summary of Thesis

Chapter 2, the literature review, discusses two complementary paradigms for building distributed systems, the first of which is service-oriented architectures. The broad topic of service-oriented architectures are introduced focusing on the web services approach, followed by a discussion of workflow technologies, the Semantic Web, Grid computing and its application, e-Science. Scientific workflow is then specifically discussed in detail, presenting an overview of the current state-of-the-art scientific workflow composition tools: myGrid, Imperial College e-Science Networked Infrastructure (ICENI), Kepler and Triana. The second paradigm is multiagent systems, our discussion here is focused on how to build distributed systems from a number of autonomous, self interested agents, specifically addressing techniques to build systems from the bottom-up (smart agents) or from the top-down (smart coordination). Highlighted in our discussion are the Electronic Institutions framework and the concept of interaction protocols.

This thesis has worked closely with a number of active Grid projects, focused on Virtual Observatory engineering, namely: AstroGrid and the LSST project. Chapter 3 introduces the broad application domain of Virtual Observatory technology, specifically the architecture of AstroGrid. The remainder of the Chapter introduces two workflow scenarios taken from AstroGrid, focused on batch processing and knowledge acquisition. Based on the review of existing scientific workflow systems and analysis of motivating workflow scenarios the remainder of the Chapter derives the core requirements of scientific workflow.

Chapter 4 presents a further workflow scenario which has been jointly derived with the LSST project, centring around runtime coordination and data classification. This scenario acts as a counterexample of coordination which is difficult or impossible to achieve through existing service composition techniques. Through our analysis of this scenario it is apparent that an extra set of requirements are necessary, requiring flexible, dynamic, runtime composition of services.

Chapter 5 presents the MultiAgent Service Composition (MASC) language, which is an agent-based solution to the service composition problem. Our approach is founded on the concept of interaction protocols. Here the formal syntax is discussed in detail, highlighting why certain choices were made and providing simple examples of use where necessary.

A prototype implementation framework is discussed in Chapter 6. The Zorro framework serves as a full implementation of the MASC language and serves as an decentralised, peer-to-peer, agent-based service composition tool, allowing scientific workflows to be represented and enacted by describing an e-Science experiment as an interaction protocol.

Chapter 7 ties together all of the separate sections of the thesis, demonstrating how an agent-based approach to service composition can solve the motivating set of workflow scenarios and meet the requirements of scientific workflow (derived throughout this thesis). The coordination-oriented programming methodology is introduced which serves as a guideline for constructing workflows through various levels of abstraction using an agent-based approach. The methodology, language and framework are then applied to the batch processing, knowledge acquisition and runtime coordination scenarios, providing a full implementation in the MASC formal syntax and an executable XML specification. The Chapter concludes by addressing how the requirements of scientific workflow have been met and discuses the advantages and disadvantages of an agent-based approach compared to existing service composition techniques. Finally, Chapter 8, the conclusion reiterates the points made throughout the thesis, discussing further work and avenues for research.

Chapter 2

Literature Review

This Chapter presents an overview of two independent fields of distributed systems research, namely: service-oriented architectures and the development of multiagent systems. Service-oriented architectures are emerging as the de-facto standard method of deploying application code over a network. Section 2.1 introduces the concept, which Section 2.1.1 describes the simple, vanilla web service standards which allow application code to be cleanly exposed to a network. Section 2.1.2 discusses how to compose these simple services, allowing more complex coordination, known as workflow. While Section 2.1.3 discusses the Semantic Web, an extension of the current web which allows information to be given well-defined meaning, better enabling computers and people to work in cooperation. Finally in Section 2.1.4 the application of these concepts to the scientific community (also labelled as e-Science) through Grid infrastructures is discussed.

A common problem of the Grid community is composing multiple distributed, heterogenous resources into computational e-Science experiments, also known as *scientific workflow*. Scientific workflows have an overlapping set of requirements with workflows found in the Business Process Modelling domain, but it is also true that they have an additional set of requirements, and therefore need consideration separately. Section 2.2 discusses the state-of-the-art in scientific workflow systems: myGrid, ICENI and Kepler, along with some other related projects.

Multiagent systems provide an alterative paradigm for building complex distributed systems and address a fundamentally different set of problems to those of pure system building, such as a service-oriented architecture approach. The multiagent systems community's focus lies with creating *autonomous*, *flexible* software components which can operate in *open*, *dynamic* and *uncertain* environments. Section 2.3 introduces the notion of multiagent systems, Section 2.3.1 discusses how to build intelligent agents from the bottom-up and Section 2.3.3 discusses how to build communities of interacting agents from the top-down. The concept of shared interaction protocols are introduced in Section 2.3.4 with a discussion of the Electronic Institutions framework in Section 2.3.5. To conclude the Chapter, Section 2.4 presents a discussion of fusing these disjoint camps of distributed systems research.

2.1 Service-Oriented Architectures

A service-oriented architecture is an information technology approach or strategy in which applications make use of (or rely on) services available in a network, such as the World Wide Web [45]. Implementing a service-oriented architecture involves developing applications that use existing services, making applications available as services or both. A service provides a set of functionality. This can be a single discrete function, such as converting between two currencies, or it can be composed from a set of inter-connected functions, such as the process of reserving a seat on a flight. Multiple services can be glued together to perform more complex operations.

Service-oriented architectures are 'loosely coupled'. This means that the client of a service is essentially independent from the service itself. When a client (which can be another service) makes an invocation on a remote service, it does not need to concern itself with the inner workings (for example, what language it is written in) to take advantage of its functionality. The service can be treated as a black box, communication takes place through a well defined interface, and the processing is left up to the service implementation. This means that if the implementation is changed or updated the client can call the service in the same way (providing the interface stays the same).

There are many reasons for choosing a service-oriented approach to designing software systems. They allow the software engineer to *re-use* existing code. By simply wrapping existing code in a standard interface language, legacy components can be easily integrated into newer systems. Systems are more *inter-operable*, as standard interfaces and methods of communication are defined. Loosely coupled services are often more *flexible* than traditional tightly coupled applications. In a tightly coupled architecture, the different components are bound to one another, sharing semantics, libraries and often state; making it difficult to evolve the application. As services are independent from one another they offer a greater of flexility and *scalability* for evolving applications.

The concept of service-oriented architectures is not a new one. There have been many different architectures which expose software components through standard interfaces, allowing them to be composed into larger applications. Earlier architectures include Java RMI, CORBA, and DCOM [18]. However non of these standards have been so widely adopted as the web services approach to service-oriented architecture.

2.1.1 Web Services

The web as we know it today started out supporting human interactions with textual data and graphics. There are many common uses for the internet, namely reading the news, looking up stock quotes etc. However this text-based web does not support software interactions very well. A more efficient method was needed, which allowed applications to interact directly with one another, automatically executing instructions that would otherwise have to be entered manually though a browser. Web services are a distributed computing platform targeted at the Web. They define a standard way of performing program-to-program communication. Web services can tie together any application, operating system, data store, programming language, and device to any other. Web services employ a number of standards which enable this inter-operability, illustrated by the stack diagram presented by Figure 2.1.

The web services core standards are: XML, WSDL and SOAP, which will now each be discussed in turn:

- **XML:** The Extensible Markup Language [63] has become the de-facto standard for describing data to be exchanged over the web. XML is a markup language, and allows the contents of a document to be described with a set of elements. An XML document is typically associated with an *XML Schema* which describes its grammar rules. These grammar rules define which elements are allowed in the document, the structure of the elements, data expected inside the elements etc.

- **WSDL:** The Web Services Definition Language [25] defines the interface to the web service so that a client application can communicate and invoke the ser-

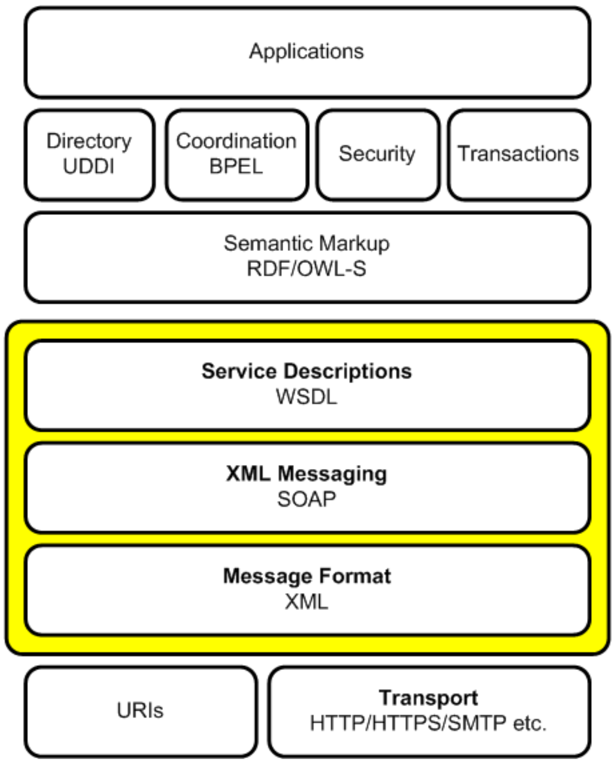


Figure 2.1: Service-oriented architecture stack overview

vice. A WSDL document describes a web service as a collection of abstract items called *ports* or *endpoints*. A WSDL document also defines the actions performed by a web service and the data transmitted to these actions in an abstract way. Actions are represented by *operations*, and data is represented by *messages*. A collection of related operations is known as a *port-type*. A port type constitutes the collection of actions offered by a web service. What turns a WSDL description from abstract to concrete is a *binding*. A binding specifies the network protocol and message format specifications for a particular port type. A port is defined by associating a network address with a binding. If a client locates a WSDL document and finds the binding and network address for each port, it can call the service's operations according to the specified protocol and message format.

- **SOAP:** The Simple Object Access Protocol [40] is an XML based protocol for exchanging information in a distributed environment. It is the plumbing of the web services toolkit. SOAP is an extension of the Hyper Text Transport Protocol (HTTP) that supports XML messaging.

These core web service standards are widely adopted by both industry and academia and have become the de-facto standard way of performing distributed remote procedure calls. As illustrated by Figure 2.2 web services can wrap any back-end system (such as: .NET, J2EE, CORBA and legacy code etc.) presenting it to the network through a standard interface written in WSDL. Web services interfaces receive a message (formatted using XML) from the networked environment, transform the XML message into a format understood by a particular back-end software system and optionally return a reply message. The underlying software implementation of web services can be created using any programming language, operating system or middleware.

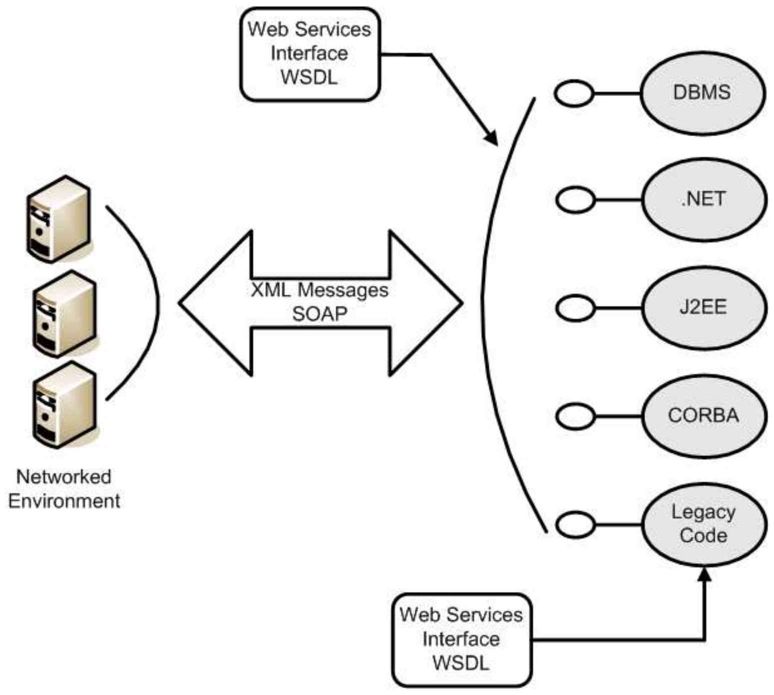


Figure 2.2: Putting web service standards together

More complex behaviour can then be built on top of this relatively simple set of core standards, see the top of Figure 2.1. Standards which allow for example: Semantic markup of web service descriptions, choreography of web services into more complex coordination (such as workflow), web service transactions, web services security for encrypting XML messages and directory services which allow service advertising and discovery etc. Many of these standards are competing for the same space and some are currently at the specification stage, however implementations of these standards allow application developers to take advantage of the complex functionality. To expand on these ideas the following Sections discuss the Semantic Web, how to compose multiple services into a workflow, and Grid computing in relation to its application to the scientific domain, e-Science.

2.1.2 Composing Service-Oriented Architectures

Web services in their vanilla form provide a simple solution to a simple problem. The problem of distributed remote procedure calls with a standard set of interfaces. Things become more complex when a group of services need to coordinate together to achieve a shared task or goal. This coordination is often achieved through the use of workflow technologies. As defined by the Workflow Management Coalition [31], a workflow is the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant (a resource; human or machine) to another for action, according to a set of procedural rules.

Business workflow technology dates back as early as the mid 1970's and the first attempts to automate business processes were part of the office automation prototypes developed at Xerox PARC. The initial idea was to reduce the complexity of the user's interface to the office information system, control the flow of information, and enhance the overall efficiency of the office [24]. This movement truly gained ground in the 1990's under different names, including business process modelling and business process engineering.

All workflow systems contain a number of generic components which interact in a defined set of ways; different products will exhibit different levels of capability within these generic components. The workflow reference model, illustrated in Figure 2.3 has been developed as a generic framework to show the interactions between the major components and interfaces of workflow systems.

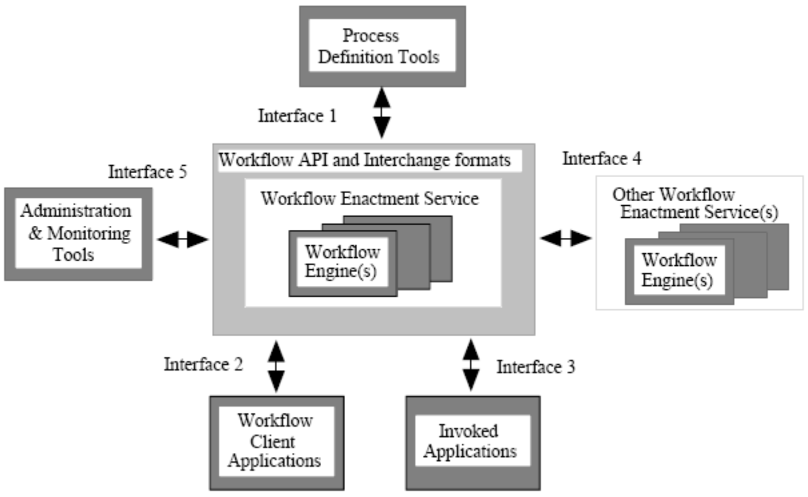

Process Definition Tools
Interface 1
Workflow API and Interchange formats
Interface 4
Interface 5
Administration & Monitoring Tools
Workflow Enactment Service
Workflow Engine(s)
Other Workflow Enactment Service(s)
Workflow Engine(s)
Interface 2
Interface 3
Workflow Client Applications
Invoked Applications

Figure 2.3: Workflow reference model - components & interfaces

The *workflow enactment service* is a software service that consists of one or more workflow engines in order to create, manage and execute software instances. The *workflow engine* provides the run-time execution environment for a workflow instance. These two components interact with a number of external components though a set of well defined interfaces, marked on Figure 2.3:

- **Interface 1:** A variety of *process definition tools* can be used to model the business process; the output of which is a process definition which can be interpreted at run-time by the workflow enactment service. This is achieved by passing a complete process definition or subset through the process definition import/export interface.

- **Interface 2:** Connects the running workflow instance to a client application. These applications will typically want to: create/terminate workflows, suspend a running workflow, retrieve process data and provide user driven input to the workflow.

- **Interface 3:** Defines how the workflow enactment service can interact with external applications/services; such as databases, visualisation software and web services.

- **Interface 4:** Allows a particular workflow enactment service to communicate with another enactment service, which has a differing process model and execution semantics. Translation between the two models may be necessary.

- **Interface 5:** Connects the workflow enactment service to external tools for the administration and monitoring of the workflow. Typical functions of these tools will include: checking the process state, adding/deleting users privileges, and modifying running processes.

Most existing workflow systems adhere to the workflow reference model, however there are many competing process description languages. This space is crowded with many organisations, each wanting their own standard to be adopted by the community as the way of coordinating distributed resources over the internet. The current front runner is BPEL (Business Process Execution Language) [7] for web services. However there are many other standards which share the space, such as WS-Coordination [3] and the Web Services Composite Application Framework (WS-CAF) [15]. A comparative study illustrating the differences between these languages is presented in [65].

2.1.3 The Semantic Web

The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation [11]. The web can reach its full potential if both humans and machines can understand and process the available information. Currently this is not possible as the web is based primarily on documents written in the HyperText Markup Language (HTML) which contains no facilities for expressing semantic information. The Semantic Web aims to addresses this shortcoming using the descriptive technologies Resource Description Framework (RDF), Web Ontology Language (OWL) and the Extensible Markup Language (XML) [8]. When combined, these technologies provide descriptions that supplement or replace the content of web documents. This semantically marked up web content then becomes machine-readable, thereby facilitating automated information retrieval by computers. The Semantic Web comprises of a number of standards and tools:

- **XML and XML Schema:** As described by Section 2.1.1 XML provides a surface syntax for structuring documents, however it imposes no semantic constraints on the meaning of these documents. XML Schema is used to express a set of rules which define the legal building blocks of an XML document, typically expressed in terms of constraints on the structure and content of documents.
- **RDF and RDF Schema:** The Resource Description Framework (RDF) is based around the concept of making statements about resources (objects) in the form of subject-predicate-object expressions called RDF triples. RDF Schema is a vocabulary for describing properties and classes of RDF resources.
- **OWL:** The Web Ontology Language (OWL) adds another layer of vocabulary for describing properties of classes, for example relations between classes, cardinality etc.

These technologies are important for the future implementation of the Semantic Web, however they have not yet been widely adopted by the academic or industrial communities. In relation to our discussion of service-oriented architectures, OWL-S [39] is a web service ontology for describing service descriptions. This language supplies web service providers with a core set of markup language constructs for describing the properties and capabilities of their web services in unambiguous, computer-interpretable form. The ultimate aim of such a language is that web services can be composed automatically from the attached semantic information by a user agent.

2.1.4 Grid Computing

In a world where communication is nearly free, when solving problems we are not restricted to the use of local resources. Computationally intense jobs can be executed on the collective resources of research and industrial partners, simulations can be run remotely rather than locally installing software, remote data can be accessed and processed directly. The problem is that these resources are often owned by different organisations, have differing security policies, run different software etc. These are standard problems in the distributed systems community, and so just having network access to these resources is simply not enough to tie everything together.

The term *'Grid'* refers to a new infrastructure that builds on today's Internet and Web to enable and exploit large-scale sharing of resources within distributed, often loosely coordinated groups, commonly termed *virtual organisations* [2]. Grid computing has attracted a great deal of interest and funding firstly from the computer science community, but also from the application of this computing research to problems in the engineering and the physical sciences.

Much of the computer science research has focused on the development of Grid middleware, in order to provide a standard and uniform mechanism for critical tasks in distributed systems. These tasks include managing services on remote computers, 'single sign on' procedures, security polices management, service discovery, transferring large amounts of data and forming large scale distributed virtual communities from a group of heterogenous components. This set of standards and mechanisms allow users to easily access this universal source of computing power for the purpose of solving problems in science (e-Science) and business (e-Business).

Up until recently the de-facto standard Grid toolkit was the Global Grid Forum's Open Grid Services Infrastructure (OGSI) [64]. This specification defined mechanisms for creating, managing, and exchanging information among entities called *Grid services*. Succinctly, a Grid service is a Web service that conforms to a set of conventions (interfaces and behaviors) that define how a client interacts with a Grid service. Grid services built on the current web service technology by extending WSDL and XML Schema definitions to incorporate amongst others, the concept of stateful web services.

This notion of state is something that web services specifications did not address and was considered necessary to provide for the controlled management of the distributed and often long-lived state that is commonly required in sophisticated distributed applications.

Although the OGSI addressed some important issues in long-lived distributed computations, the world had adopted the service-oriented architecture offered by the web services community. The main problem was that these two worlds were not interoperable with one another. It was highly undesirable to reach deadlock between the ever growing deployment of web services and the notion of state, for long-lived distributed computations found in OGSI. This led to a convergence of interest between the web services and Grid communities, resulting in the Web Services Resource Framework (WS-Resource) [19].

The WS-Resource specification was proposed to address the relationship between stateful resources and web services. It consists of a group of specifications which allow a programmer to declare and implement the association between a web service and one or more stateful resources. Importantly the framework introduces support for stateful resources without compromising the ability to implement web services as stateless message processes, meaning the two are completely inter-operable. This latest specification fills the void between the web services and Grid communities, and can be viewed as a re-factoring of the concepts addressed by the OGSI in a manner which exploits the recent developments in web services architecture.

2.2 Scientific Workflow Systems

As the Sciences become increasingly data and information driven, scientists are sharing their data and computational resources. As a direct result of this, new knowledge is acquired from analysing existing data; which would not have been previously so readily available. This information explosion has helped to shape new multi-disciplinary fields [38] such as bioinformatics, geoinformatics and neurofinformatics. The Grid is the infrastructure and machinery which enables e-Science; however current Grid software is still too complex for most scientists to exploit. Instead they require higher level tools which enable them to plug together problem solving components, in order to falsify a scientific hypothesis. A scientific workflow attempts to capture a series of analytical steps which describe the design process of these experiments. There is an increased level of interest [23, 21, 75, 20] within this domain and the problem of applying the formal concepts of workflow to the scientific community is only just starting to be addressed.

A *scientific workflow system* is an environment which combines scientific data management, analysis, simulation, and visualisation tasks in order to aid the scientific discovery process. This Section will review the state-of-the-art in scientific workflow systems from a range of application domains.

2.2.1 myGrid

Bioinformaticians conducting computation experiments would traditionally have to chain together database searches and analytical tools, using complex scripts as glue to overcome the incompatibilities between applications. Information would often need to be formatted to application-specific file formats and then passed through a selection of scientific applications and filters, which would yield a handful of results, or generate new data. These new data would in turn require reformatting and passing through further services. A scientist working using this methodology would have to transfer the results between services by hand, making note of them, re-keying the information into a new interface. This is a highly inefficient way of conducting science over the web.

myGrid [53] is a UK e-Science project which provides a set of transparent, loosely-coupled, semantically-enabled middle-ware to support scientists performing data intensive in-silico [77] experiments on distributed resources. An in-silico experiment is a procedure involving the use of local and remote resources in order for a scientist to test a hypothesis, derive a summary, search for patterns or demonstrate a known fact [52]. myGrid is implemented as a service-oriented architecture, based on web service standards. It is not designed to replace projects such as Globus [28], but rather add an extra layer of functionality above these frameworks. It is a working tool for scientists to use now and provides facilities for a number of different kinds of users, illustrated in Figure 2.4. myGrid can be presented with varying levels of abstraction from the complex wiring of web services for a Grid engineer to a high level abstract view for a non IT specialist.

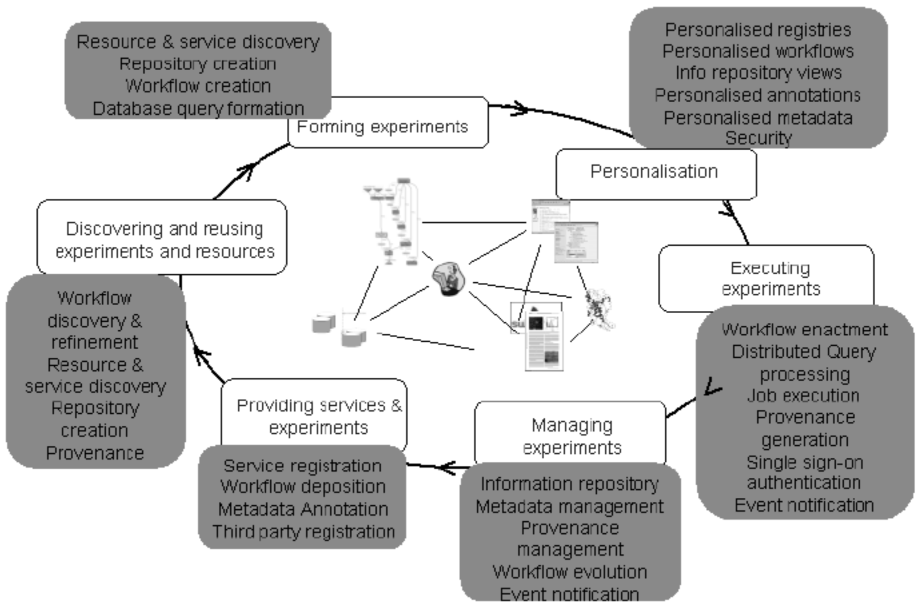


Figure 2.4: The life-cycle of an in-silico experiment [52]

Current workflow languages were deemed unsuitable for composing services within the scientific domain. Firstly because most of the standards were constantly in flux and secondly web services standards did not provide the correct level of abstraction for bioinformaticians. This led to the creation of a new language, the Simple Conceptual Unified Flow Language or SCUFL [44] for short. The SCUFL language is a high level XML-based conceptual language, in which each processing step of the workflow language represents one atomic task. It is a declarative language where the user describes what is to be done rather than how the task is performed. A user can construct a workflow in the SCUFL language by using the three main entities:

- **Processors:** Act as black box of computation. A processor consumes a set of input data and in return produces a set of output data. A processor is assigned a name and a set of input and output ports, which are named uniquely and typed within the scope of the processor. An execution status is assigned which is either: initialising, running or complete. The main types of processors [43] are: A WSDL Type definition (external web service), a SOAPLAB type (allowing command line tools to be exposed as a web service), a Talisman Type, a Nested workflow, a String constant or a Local Processor Type (calls to local class definitions).

- **Data links:** Indicate the flow of data through the workflow system, between the data source and the data sink. A data source can be defined as a processor output or a workflow input and a data sink can be a processor input or a workflow output. Each data sink will receive the same value if there are multiple links from a data source.

- **Coordination constraints:** Can be placed on two processors in order to enforce control flow over the system. This is used when certain stages of the workflow must be executed in a set order, yet there is no direct data dependency between them. A workflow can be constructed more often than not without the use of coordination constraints.

Using workflows as part of a scientific process often requires *provenance* [35] data to be kept about the activities performed during the workflow. Provenance data attempts to capture which person conducted the experiment (who); the materials and methods used in the experiment (what and how); the purpose of running the experiment (why) as well as the results and conclusions of the experiment (what). This includes data such as, when the workflow was begun, how long it took, which service instances were used, the input-output relationships between the workflow components, any intermediate data, which data were used, and the results from the workflow. In the myGrid system provenance logs are generated in the form of XML files when the enactment of the workflow begins. The system also allows storage of annotations regarding the hypothesis of the experiment along with any thoughts and opinions of the scientist. Provenance data is an important aspect of bioinformatics or any scientific experiment process; often if the same experiment is run at different times, different results are produced. Using provenance data it is possible for a scientist to trace the audit trail of previous experiments in order to add to their own experimental design: Looking at what worked, what didn't work, how it worked etc. The myGrid System allows provenance documents to be linked together enabling e-Scientists to browse and annotate them on the fly: this is the fundamental concept of the 'web of Science' [30], proposed by Hendler. myGrid offers a number of other standard services, including the notification service [36] for asynchronous delivery of messages.

The Taverna workbench [43] is the resulting implementation of this research. This tool which allows users to construct analysis workflows from components on both remote and local machines, run these workflows on a set of data and visualise the results. Within this tool is an application called the SCUFL workbench which allows scientists to write workflows in a visual format without directly using the SCUFL language.

The Taverna workbench uses FreeFluo [74] as the enactment engine, which is a web services orchestration tool: currently this tool supports a subset of WSFL and SCUFL.

2.2.2 ICENI

Imperial College e-Science Networked Infrastructure (ICENI) [41] Architecture is a service-oriented integrated Grid middleware that provides an augmented component programming model in order to aid the application developer in constructing Grid applications. An execution infrastructure is provided, which exposes compute, storage and software resources as services with well defined conditions of when and by whom these resources may be used. It is essentially a framework that enables a user to construct an application from a number of software components in a repository, based on a scientific goal. The framework then uses this component metadata to build a run-time representation, which is used to find an optimal mapping of the application to the available Grid resources at run-time.

The ICENI component framework is based upon two key principles: separation of concerns, and the utilisation of information at all stages of a computation. By capturing metadata regarding the component from its definition, its assembly into an application, through to its deployment onto distributed resources, we can influence the placement of a component network so as to maximise user and resource provider criteria.

A component is described by a set of documents that capture its meaning, behaviour and implementation respectively. This separation isolates meaning, based upon typed dataflow between components, from the associating flow of control. User construction of an application relies exclusively upon the information in the meaning level. Each document is defined in terms of a different XML Schema. A component has a set of ports through which all communication flows. Each XML document describes the same port, with differing levels of abstraction.

- **Meaning:** Describes the composability of the component and the flow of data between multiple components. The component consists of a set of ports. Each port represents the production or consumption of data from the system. At the meaning level a port has an associated dataflow, *in*, *out* or *exchange*. An inport, represents the consumption of data, an outport represents production of data and, finally, an exchange represents a port which performs both. An abstract data type (identifying the type produced or consumed) is associated with an inport or outport, while an exchange posses two types, indicating the flow in and out of the port. A port at this level is defined using the Component Definition Language (CDL).

- **Behaviour:** Captures how the data are passed from one component to another, and what dependencies exist between the dataflow relations described in the components meaning. It is described using the Behaviour Definition Language (BDL): each port described using BDL must map to a port defined using CDL.
- **Implementation:** Described by using the Implementation Definition Language (IDL), defines concrete data types, including the precise format of the data for all the components ports. This level also possesses metadata about the components performance characteristics along with platform specific requirements and settings.

Each instance of a component has a single Meaning, Behaviour and Implementation document. The associations between the files are illustrated in figure 2.5. However a single meaning may have multiple behaviours and a single behaviour may have multiple implementations. A user is *only interested* in the component's meaning, while selection based on a component's behaviour and implementation are handled by the Grid middleware.

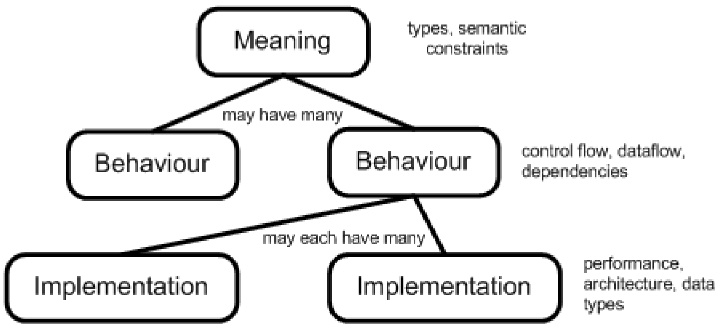

Meaning
types, semantic constraints
may have many
Behaviour
Behaviour
control flow, dataflow, dependencies
may each have many
Implementation
Implementation
performance, architecture, data types

Figure 2.5: Levels of semantic information in ICENI

A user of ICENI constructs an application using the information presented at the meaning level. An application is constructed by defining a set of component instances along with a set of links (which is defined as an ordered pair of component ports). The links represent channels of data flow from one component instance to another and each link must connect only two ports. The links must satisfy two criteria: firstly, the abstract data types must be the same and the dataflow directions must be compatible (e.g. an outport must be connected to an inport etc.) The links represent channels of data flowing between concurrently exiting components, control flow issues are therefore hidden from the end user. It is then left up to the middleware to select the components based on behaviour and implementation.

Figure 2.6 illustrates an overview of the ICENI system architecture. Application construction is aided by a visual programming language using the ICENI visual composition tool. Once an application is constructed and the links between the components have been defined, an Application Description Document is generated. This XML document is passed to the scheduler which has a number of tasks it must perform. Firstly, the middleware takes the user's abstract choice based on Meaning and must choose between the various implementations, each with their associated behaviour. The middleware must select the optimal implementation for the user's chosen abstractions. Once the resources have been selected the scheduler creates new component instances and establishes links between these instances.

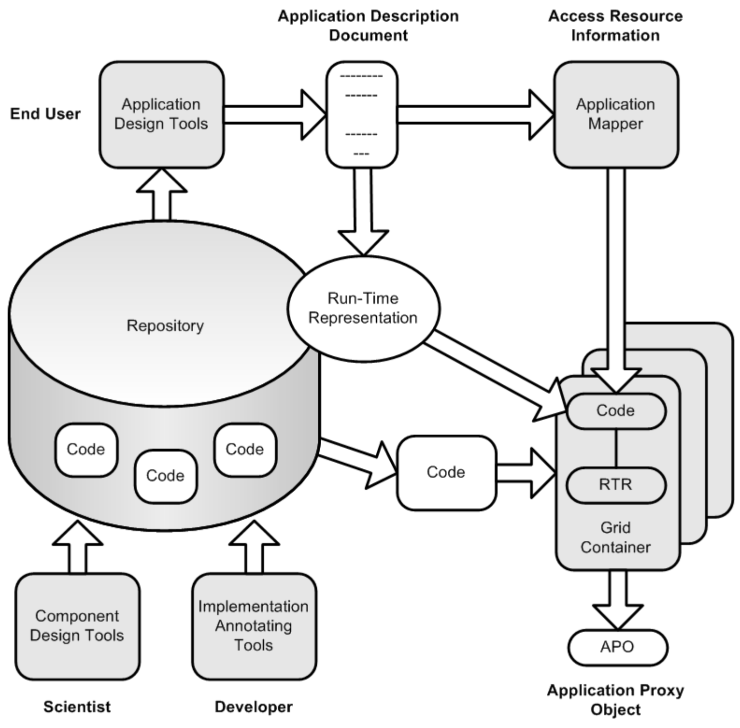


Figure 2.6: ICENI architecture overview

The XML that describes the component is used to construct bindings that allow the component to interact with the middleware and hence the Grid. Either WSDL or Java interfaces can be generated from a component's definition, which itself is defined using

CDL. Once the component is bound to an interface it can be deployed as a resource on the Grid.

ICENI has two clearly defined domains. Firstly a private administration domain, which is used to manage resources within an organisation. Secondly a public domain that exposes the private resources as services, making them available to the wider community. In between the private and public domain sits a domain manager. Its job is to add an access control policy to a resource in the private domain and expose it as a service in the public domain. This means that the same resource can be tagged with different usage policies for different computational communities. When a request comes in to access a resource the domain manager validates the request. A contract of Service Level Agreement (SLC) is defined for each resource, this states who may access the service, for how long etc.

2.2.3 Kepler

Kepler [38, 6], is an open source scientific workflow engine with contributors from a range of application-oriented research projects, for example SEEK [42]. The first thing to note about Kepler is that it is built upon the Ptolemy II system [55] based at the University of California at Berkeley. The Ptolemy II System is a mature dataflow oriented workflow architecture and is the only available system which allows different execution models to be plugged into the same workflow.

The Ptolemy II System introduces a number of basic blocks which form ‘actor-oriented’ workflow modelling, illustrated in figure 2.7. The most basic component in the system is an *actor*. An actor is simply an independent unit of computation (such as a web service, database call etc.) which consumes *data-tokens* from a set of *inports* and produces data-tokens to a set *outports*. These ports provide the communication interface to other actors in the workflow. A group of actors can then be ‘wired’ together by introducing a mapping of outports to inports. An actor can consist of a sub-group of interconnected actors, allowing hierarchical workflows to be supported: this is known as a *composite actor*. In addition to the connection of ports to form the dataflow model, control flow can be enforced through the use of branching and looping.

The component communication (dataflow) concerns are separated from the overall workflow coordination which is defined in a separate component called a *director*.

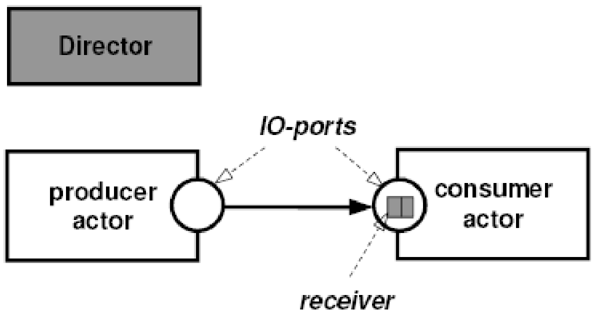


Figure 2.7: The Kepler actor model

The execution model defined by the director is known as the *model of computation*. This separation of concerns means that once a workflow model is constructed it can be run with different execution semantics; defined within the director.

Kepler has a number of built in actors, providing facilities for: prototyping workflows, executing web and grid services, distributed job execution, database access etc. Inherited from the Ptolemy II System are a number of built in models of computation that the directors can enforce. These include the Synchronous Dataflow, Process Network, Continuous Time and Discrete Event models. Additional models of computation can also be introduced into the framework, allowing user-defined execution semantics.

The modelling concepts introduced do not have to be bound to a particular group of types at design time. The Kepler system builds upon the actor-oriented modelling approach and introduces type definitions to be represented in a number of ways:

- **Structural types:** Define the allowed set of values that a port can consume or produce. The language used to describe the structural type of a port could be an XML schema, DTD or programming language type system for example. When using XML schema, the structural data type of a port is a concrete XML schema type, such as `xsd:string`.

- **Semantic types:** Allow the user to define a concept expression over a language used to model ontologies, such as a description logic. As an example, a user might define a semantic type which states that only data tokens which describe a species of mammal can be placed into the input port of the actor.

- **Hybrid types:** Allow structural and semantic types to be explicitly linked through the use of *hybridization constraints*. These constraints can be exploited in a number of ways, for example to infer (partial) structurally mappings between structurally incompatible (but semantically compatible) workflow components.

Constraining the port definitions of an actor by defining the type definition of its input and output ports allows the underlying workflow system to check (at design time) that the connections between ports are consistent. In this way faulty links due to type mismatches can be identified and corrected before the workflow is executed. Structural and Semantic types are separate concerns and the user can choose to type a port with either or both of these type definitions, depending on the information available at the design stage: these definitions can always be altered later. This separation of the data modelling (structural type) and conceptual modelling (semantic type) allows them to be independently validated and offers a number of benefits for scientific workflow and component reuse. A formal overview of the Kepler system can be found in [14].

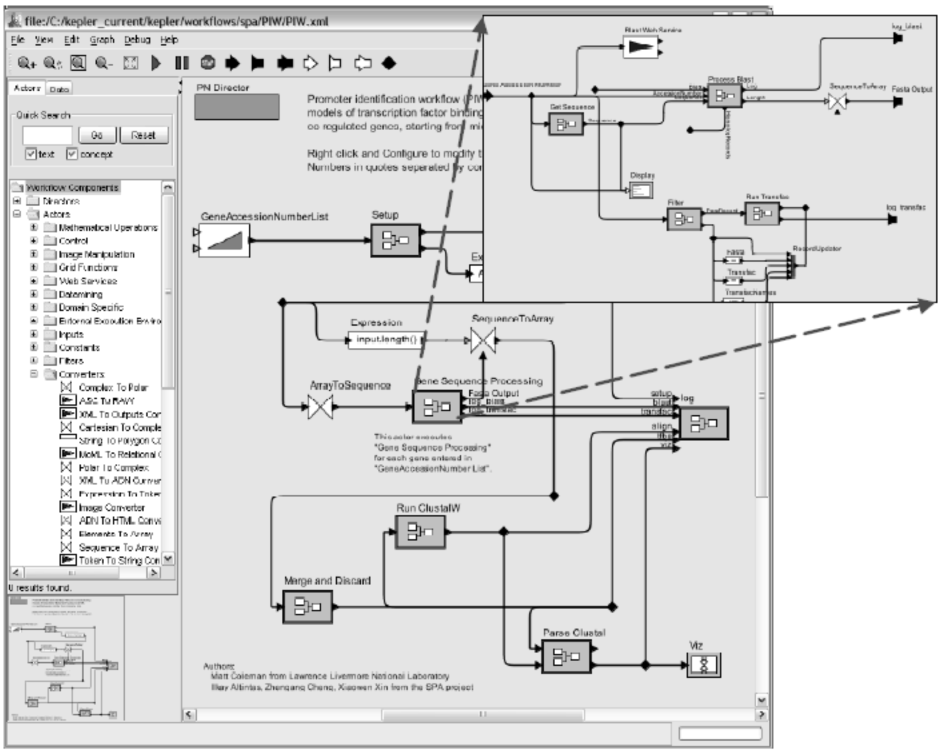


Figure 2.8: The Kepler GUI [71]

In summary, Kepler offers a highly flexible scientific workflow execution environment with well supported tools and visualisation software. Different ready made actors and models of computation can simply be plugged into the workflow to achieve the desired behaviour. The GUI is intuitive, allowing workflows to be viewed at differing layers of abstraction; depending on whether the user is a scientist or Grid engineer.

2.2.4 Related Projects

- **Triana** [54] is an open source problem solving environment, designed as a flexible software development kit and is intended to be used in many different scenarios and many different levels of abstraction. Triana is a test application for the GridLab project [5], a set of middleware for the creation of generic Grid applications. The toolkit allows users to compose workflows graphically using a dataflow model on their local machine and distribute this load over a connected Peer-to-Peer network.

- **eStar** [16] is a software project which aims to develop an intelligent robotic telescope network. It is a joint collaboration between the Astrophysics Research Institute at Liverpool John Moores University, the Astrophysics Research Group at the University of Exeter and the Joint Astronomy Centre (JAC) in Hawaii. This project addresses the application of ‘intelligent’ agents to a network of robotic telescopes. An intelligent agent resides on a user’s local machine and can both request observations and receive data from telescopes which is potentially of interest to the user. The user agent interacts with discovery nodes on the network via Grid middleware. Discovery nodes are a collection of sub-systems (telescope, database, agent etc.) which can receive observation requests from an agent and through a series of interactions produce some astronomical data.

  It is being deployed on the United Kingdom Infrared Telescope’s Wide Field Camera (WFCAM) and will cross correlate this output with the set of known objects taken from pre-existing survey databases. The science aim of developing this robotic telescope network is to aid the detection of transient and moving objects in the sky, enabling agents to rapidly compare output data from WFCAM to existing objects in order to schedule follow up observations on these newly detected objects. It is necessary to schedule a follow up observation as soon as possible in order to avoid loss to potential time-sensitive results.

2.3 Agent Oriented Software Engineering

Multiagent systems are a relatively new field of Artificial Intelligence (AI) and currently a highly active area of research. This field brings together researchers from hugely diverse areas of study, ranging from computer science to social science.

Agent is a contested term, principally because different domains consider different traits of agents to be more important than others; each having their own definition of what they mean by the term agent. Some applications for example require that agents have the ability to learn, but for other applications this is an undesirable trait. There is however, a common thread of consensuses for the term agent throughout most application domains.

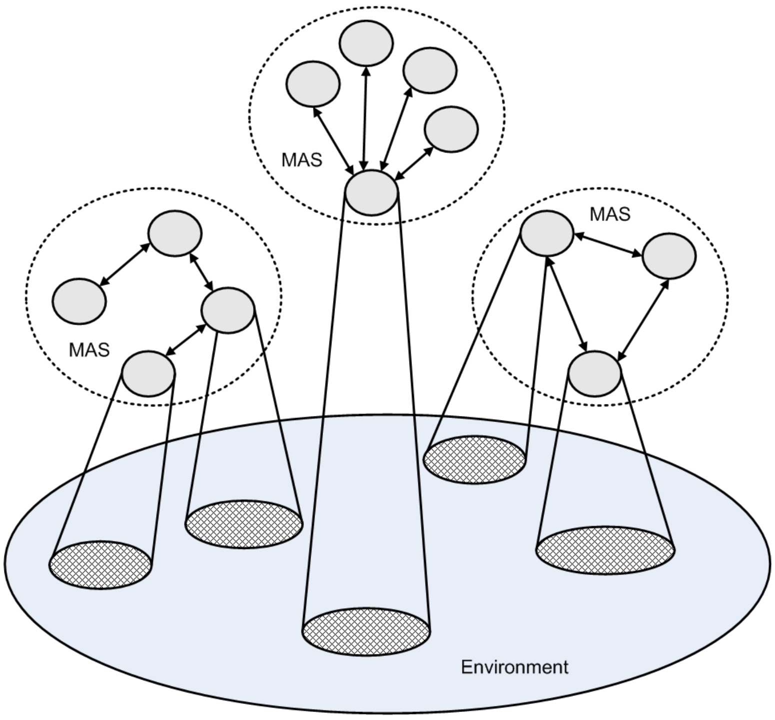


Figure 2.9: Multiagent systems

An agent is a computer system that is situated in an environment, and that is capable of autonomous action in this environment in order to meet its design objectives. An agent usually takes sensory input from the environment (which is assumed to be non-deterministic) and produces as output actions that affect it [73]. An agent can usually influence its environment. This means that the same action performed twice in apparently identical circumstances, might appear to have completely different effects. An agent will usually have a collection of actions that it can perform on its environment under certain circumstances. The key problem facing the autonomous agent is deciding which of its actions it should perform in order to meet its design objectives.

A multiagent system consists of a number of autonomous agents, which communicate with one another through a computer network infrastructure. Figure 2.9 illustrates this concept and shows the relationships between groups of agents and the influence they have on their surrounding environment. Individual agents will have been designed by different engineers and will therefore exhibit different behaviour through their goals, motivations and internal logic. If agents are to successfully interact with one another they will require the ability to *cooperate*, *coordinate* and *negotiate*. Multiagent systems are often viewed as a society, although the agents are autonomous and can act independently, the society lays down conventions that allow the agents to cooperate with one another to achieve a shared goal.

A good example of a multiagent system is an auction house. Agent interactions take place between an auctioneer agent and a collection of bidder agents. The aim of the auction is to allocate the item to one of the bidders. The auctioneer wants to maximise the profit of the item in hand, where as the bidders want to acquire the item at the lowest possible price. Here the laws of trade (English auction, Dutch auction etc.) are pre-defined and agents must adhere to these rules in order to successfully take part in the auction. However within these rules there is scope for these independently engineered agents to act autonomously at run-time, adopting different bidding strategies and tactics to acquire the item for the cheapest price, in order to maximise their own gain.

Many researchers are skeptical about the claims made by the multiagent systems community. Some arguments suggest that it is simply repackaged distributed and concurrent systems, artificial intelligence and game theory. As a paradigm for software engineering multiagent systems have a great deal to offer. Agents can be inherently decentralised peer-to-peer systems, compared with the traditional client-server model. They therefore exhibit improved scalability and do not suffer from the single point of failure problem. By offering a degree of autonomy to agents, the complexity of design in a multi-threaded system is drastically reduced; principally because the concurrent interactions can be left up to the agents at run-time, and not specified explicitly like the traditional top down design of distributed systems.

There are essentially two ways to build multiagent systems (illustrated in Figure 2.10)

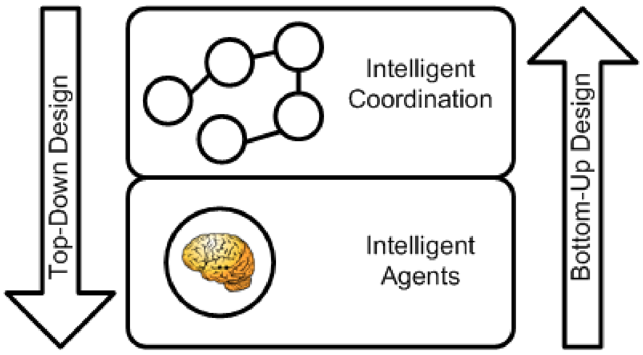


Figure 2.10: Intelligent coordination vs. intelligent agents

and agent researchers broadly fall into one of these two categories. Engineers can design systems from the bottom-up, focusing on producing *smart agents* known as the *agent-design* problem, or from the top-down, focusing on *smart coordination*, otherwise known as the *agent-society* design problem.

2.3.1 Smart Agents

Researchers interested in the agent-design problem are concerned with producing individual, intelligent agents. Concerns lie with how users might tell the agents what to do, and how agents themselves decide which actions to perform, through various types of logical reasoning:

- **Deductive reasoning agents:** Intelligent behaviour can be simulated by manipulating a symbolic representation of an environment and the desired behaviour within this environment. This is the traditional approach to building artificially intelligent systems, known as symbolic AI. Theorem proving is a technique used to create deductive reasoning agents.

- **Practical reasoning agents:** It is clear that we as humans do not use a purely logical approach to reasoning, as addressed by deductive reasoning. Practical Reasoning is concerned with decision making directed towards actions, this decision making is a consequence of weighing up often conflicting considerations for competing options. Deciding whether to catch the train or the bus is an example of practical reasoning, as it is reasoning directed towards action. Practical reasoning consists of deliberation (deciding what to do) and means-end reasoning (how to do), best known as *planning* in the AI community. Agent researchers are interested in how to use practical reasoning techniques to give agents a degree of autonomy, so they can ultimately make decisions for themselves.

2.3.2 Smart Coordination

By allowing agents to coordinate together it is not necessary to focus on engineering individual smart agents, there is a notion of shared intelligence and cooperation. There is a popular slogan in the multiagent systems community: there is no such thing as a ‘single agent system’ [73]. This illustrates the point that interacting systems are now the norm, computers are pervasive and expected to interact for even the most basic of tasks. By taking a top-down approach to the design of multiagent systems there are a number of key issues that need to be addressed:

- **Reaching agreements:** As agents are considered to be autonomous and self interested entities, it is necessary to study how they can reach mutually beneficial agreements on matters of common interest (similar to the society we live in), without a third party to dictate the terms. Negotiation scenarios will usually be governed by a protocol which lays down the common rules of encounter. Agents must adhere to this protocol in order to take part in the interaction. However an agent remains self interested, it will adopt a particular strategy which attempts to maximise its own gain. Researchers are interested in how to design such negotiation protocols, build strategies around these protocols so agents can negotiate on behalf of users and understand the process of reaching agreements through techniques, such as negotiation and argumentation.

- **Cooperation:** The focus here lies on how agents can collectively work together in order to solve a shared problem. The distinction between multiagent cooperation and traditional parallel problem solving is inherent within the term agency. The *benevolence assumption* states that agents in a system implicitly share a common goal and that there is no potential for conflict between them. The benevolence assumption however is generally not accepted when agents are interacting in an open system. Agents are engineered by different individuals and will therefore be motivated by a different set of goals. When taking part in coordination they will (as self interested entities) lean towards the outcome which maximises their own gain. As they act autonomously, decisions are not hardwired in at design time, as they traditionally are with distributed/concurrent systems. Agents must be able to dynamically make decisions within their environment at run-time. The task of cooperation is far more complex when dealing with these self interested agents, research into cooperation through norms and social laws is a popular technique to enforce control in an open system.

The remainder of this Section focuses on how to design multiagent systems from the top-down, by providing these open system with intelligent coordination mechanisms. Firstly the broad topic of agent communication is discussed, with reference to languages: FIPA-ACL and KQML. The Electronic Institutions framework is then which is a popular technique allowing structure and organisation to be imposed on an open system.

2.3.3 Agent Communication

Agents communicate with one another through message passing. As an agent is considered to be autonomous and in control of both their own state and behaviour it cannot be expected that just because you tell an agent to do something, it will necessarily complete this task. It might not be in the agents best interest, or might not be possible.

Instead agents can perform communicative actions in an attempt to influence another agent. For example when I tell a friend to ‘meet me at the pub at 7pm, and on this occasion I will be on time’. Although I am trying to influence my friend to turn up at 7pm, he is in control of his own beliefs, desires and intensions and realises from previous experience, I often run late. Hence he decides to turn up a little after 7pm instead. But by performing this communication I am attempting to change the internal state of my friend. This idea is captured within the theory of *speech acts*. Speech acts, are a certain class of natural language utterances which have the characteristic of actions, in the sense that they change the state of the world in a way representative of physical actions. Speech acts were originally explored through the work of philosopher John Austin [9]. An example of such an utterance is the declaration of war or a marriage declaration. Various types of speech act are classified into *performative verbs*, such as request, inform and promise.

A number of Agent Communication Languages (ACL) have been developed which use the theory of speech acts as a basis. The Knowledge Query and Manipulation Language (KQML) [33] was the earliest attempt. It was a DARPA funded project which specified a common format for the interchange of messages between agents. Although widely adopted this language was criticised for having no formal semantics and an under constrained set of performative types (41 in total). This resulted in different implementations adopting different performative types to mean the same thing. The Foundation for Intelligent Physical Agents (FIPA) Agent Communication Language (ACL) [1] was designed as a standard to address the short comings of KQML. FIPA-ACL has a more concrete, formal syntax and fewer, more meaningful performative types. This is currently the most widely adopted ACL in use by the multiagent systems community.

2.3.4 Interaction Protocols

An interaction protocol is essentially a collection of conventions which allow agents in an open system to interact with one another. The term *open system* means that any agent can take part in the interaction, regardless of their internal implementation details; such as the language they are programmed in, or operating system they are run on.

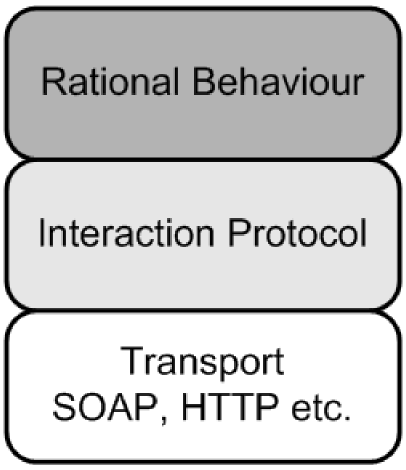


Figure 2.11: Interaction protocols

Firstly it is important to address what an interaction protocol does not define; it does not attempt to define the transport mechanism used to get messages from one agent to another, such as HTTP, SMTP or SOAP etc. These are regarded as low level programming issues and are not the concern of agent communication. Nor does it attempt to define what the agent does internally when it receives a message, such as how the agent rationalises. This is left up to the individual agent implementation and these issues can be regarded as higher up the stack. An interaction protocol sits between the transport layer and the rational layer, illustrated by Figure 2.11. An interaction protocol defines the *rules of engagement* between a group of interacting agents. Such as *if and when* an agent can communicate, and the *order* and *kind of messages* that an agent expects. A protocol is domain and situation specific, e.g. a Dutch auction protocol would be radically different to an English auction protocol.

2.3.5 Electronic Institutions

Electronic Institutions (EI) [26] are a technique used for providing structure and organisation in an open multiagent system. EIs are modelled by observing the conventions that make up human organisations. Human societies have created institutions; inside these institutions they set laws, monitor and respond to emergencies, prevent and recover from disasters etc. By modelling these conventions key issues in open multiagent systems, namely heterogeneity of agents, trust and accountability, exception handling (detection, prevention and recovery from failures) and societal change have been addressed. This allows heterogeneous agents implemented by different engineers to communicate, negotiate and cooperate with one another in a truly open system.

EI is the term given to the formal representation of these concepts and has resulted in a framework for open multiagent systems which attempts to mimic a human institution. It forces agents to interact with one another in a well-defined manner and to adhere to roles commitments and obligations. The core concepts of EI are not that dissimilar to a theatre production. An EI consists of a number of components, the most basic of these being an *agent*. Agents can be viewed as the actors, and interact with one another through *illocutions*. Each agent is required to adopt one (or more) *roles* within the institution.

Roles define standard patterns of behaviour and have *dialogic actions* associated with them. Dialogic actions are a set of operations which an agent can perform, once an agent adopts a role it can perform the dialogic actions associated with that role. Interactions between agents take place only inside *scenes*. A scene can thought of as a bounded space where agents directly interact and negotiate on a single task. A scene contains a *script*, which is a well-defined protocol (modelled as a finite state machine). This protocol contains all the possible dialogue between a set of roles. A *performative structure* is a network of scenes, it defines how and under what conditions different roles can legally move between scenes. Agents within a performative structure can participate concurrently in different scenes with different roles. Actions that agents take in the context of an institution may have consequences that either limit or enlarge its subsequent acting possibilities. The set of possible paths for an agent within the performative structure is thus defined by a set of *normative rules*.

One further thing is required to allow agents to interact, common knowledge. This is represented in the *dialogic framework*. This structure contains an ontology; which defines the common language for representing the world, communication and knowledge representation. The shared dialogic framework allows heterogonous agents to exchange knowledge with other agents. With many key concepts involved it is useful to consider an example [68] of an electronic institution which ties everything together, this is graphically represented by Figure 2.12.

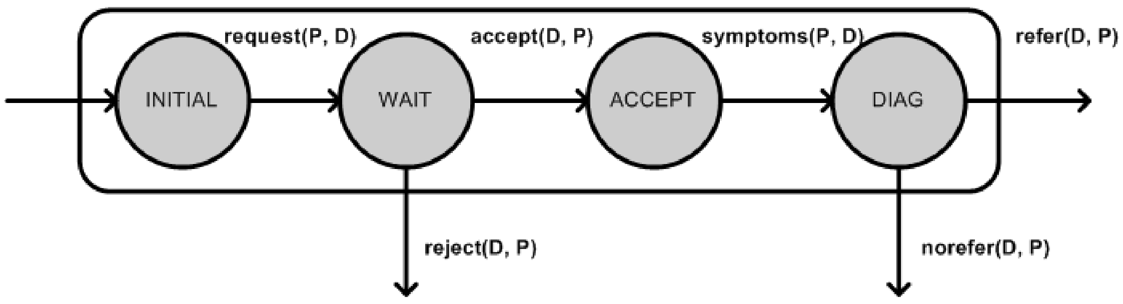


Figure 2.12: General practitioner scene

The scene represents a patient visiting a General Practitioner (GP) to obtain a diagnosis of some symptoms. There are two roles defined within this scene; the roles of doctor and patient. For convenience, we assume that all agents use the same dialogic framework (i.e. they know how to communicate) and that there are no normative rules. The scene begins with all the agents entering the `INITIAL` state. A patient agent then sends a request message to a doctor agent indicated by `request(P, D)`, where `P` is a patient and `D` is a doctor. This message is intended to represent the patient making an appointment to see a doctor. The patient then enters the `WAIT` state until an `accept(D, P)` or `reject(D, P)` message is received from the doctor. If a reject message is received, then the agent returns to the the initial state. If an acceptance is received, the agent enters the `ACCEPT` state and proceeds to send a message `symptoms(P, D)` to the doctor. The doctor then performs a diagnosis of the patient in the `DIAG` state and the result is that the agent is referred `refer(D, P)` for further diagnosis, or no-referral `norefer(D, P)` is made and the patient leaves the scene.

2.4 Chapter Conclusions

This Chapter has discussed two different approaches to the design and deployment of large-scale distributed systems. Although the concept of service-oriented architectures is not a new one, the technology has only recently reached maturity through simple, vanilla web service standards. The web service and Grid middleware is in place to provide reliable, scalable and secure access to distributed resources. The agents community, however has typically focused on creating autonomous, flexible software components. Allowing agents to operate in dynamic and uncertain environments, making decisions about interaction and cooperation at run-time. The typical defining features from each community are illustrated by Figure 2.13.

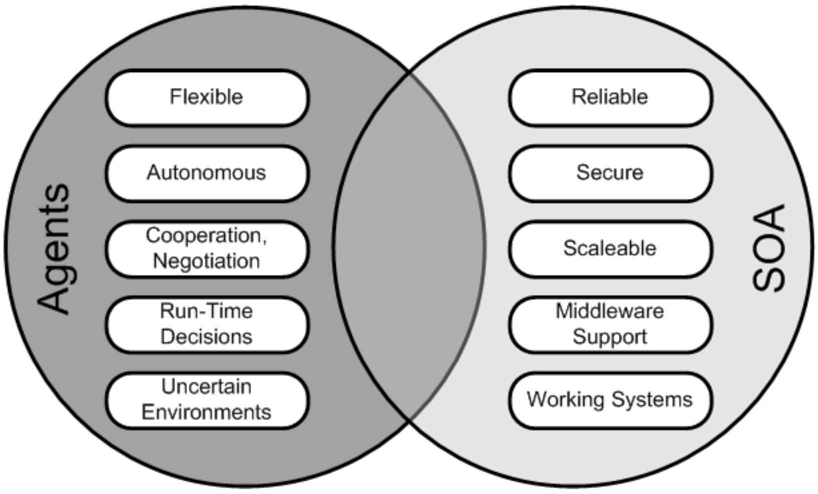


Figure 2.13: A convergence of interests

Although traditionally separate fields of research, it is clear that these two communities are starting to see a convergence of interests. The application of techniques from the multiagent systems community to service-oriented architectures is a relatively unexplored research area and in practise few steps have been taken towards the vision of fusing these two worlds of distributed systems [29]. The following Chapter builds upon the themes addressed here, further exploring the domain of scientific workflow. A set of live workflow scenarios, taken from the Virtual Observatory domain are presented in detail, these workflow scenarios demonstrate, by example the requirements of scientific workflow.

Chapter 3

Scientific Workflow Scenarios

This thesis has worked closely with a number of active Grid projects, focused on Virtual Observatory engineering, namely: AstroGrid and the Large Synoptic Survey Telescope (LSST). As a result of working with these projects a set of concrete workflow scenarios have evolved. These workflow scenarios act as a motivating factor behind this research work and are a valuable commodity in their own right. By understanding the processes behind these workflows and researching existing systems (discussed in Section 2.2) this thesis has been able to identify a set of requirements for scientific workflow, these requirements are detailed in Section 3.4.

This Chapter presents in detail two of the motivating workflow scenarios. Section 3.1 introduces the broad application domain of Virtual Observatory technology, specifically the architecture of AstroGrid. Section 3.2 discusses a batch workflow (from the AstroGrid domain) for calculating the redshift of a given area of sky. Section 3.3 details a knowledge discovery workflow (also sfrom the AstroGrid domain) which centres around retrieving and analysing data according to a scientist's hypothesis. With reference to the scenarios discussed in this Chapter and the existing system review (discussed in Section 2.2), Section 3.4 defines a set of ten core requirements of scientific workflow. Detailing how and why it differs from traditional workflow modelling. Finally, conclusions are discussed in Section 3.5. Where appropriate, workflows are described using the UML Sequence Diagram notation [61].

3.1 Virtual Observatory Technology

Breakthroughs in telescope, detector, and computer technology allow astronomical instruments to produce terabytes of images and catalogs, astronomy is facing a data explosion. The data sets produced cover the sky in multiple band widths, from gamma and X Ray, optical, infrared through to radio. With such vast quantities of data being archived, it is becoming easier to 'dial up' a piece of the sky, rather than waiting for expensive, scarce telescope time. Astronomy is being driven even further with telescopes, such as the Large-aperture Synoptic Survey Telescope (LSST) [56], capable of scanning the entire night sky in a three day period. Current estimates indicate that LSST will generate 36 gigabytes (GB) of data every 30 seconds and over a 10 hour winter night will collect up to 30 terabytes.

The software which allows the integration of astronomical resources has been slow to catch up with the ever increasing astronomy data volumes. *Virtual Observatories (VO)* are the technology frameworks which aim to fill this gap, allowing transparent access to astronomical archives, databases, analysis tools and computational services. As a direct result of collectively sharing resources through a VO, new knowledge is formed from analysing existing data; which would not have previously been so readily available. Real science has already been demonstrated using VO technologies, and as the middleware develops it will give astronomers seamless access to image and catalogue data on remote computer networks.

The International Virtual Observatory Alliance (IVOA) [66] is the standards body which ensures that all national Virtual Observatories can be integrated on a global scale. The IVOA decides upon a common set of standards and interchange formats to allow VO's to cooperate. The IVOA has grown to include 15 funded VO projects, one of which is the UK's own project: AstroGrid.

3.1.1 AstroGrid

AstroGrid [4] forms the UK's contribution to the Virtual Observatory and is a collaboration between several of the UK's leading universities. AstroGrid is funded by the Particle Physics and Astronomy Research Council to produce software within which data archives and data processing software can be accessed seamlessly by an astronomer. It is the UK's take on the Virtual Observatory concept and is a maturing system of middleware, which gives ‘workbench’ type interaction for scientists to astronomical instrumentation, services and archives.

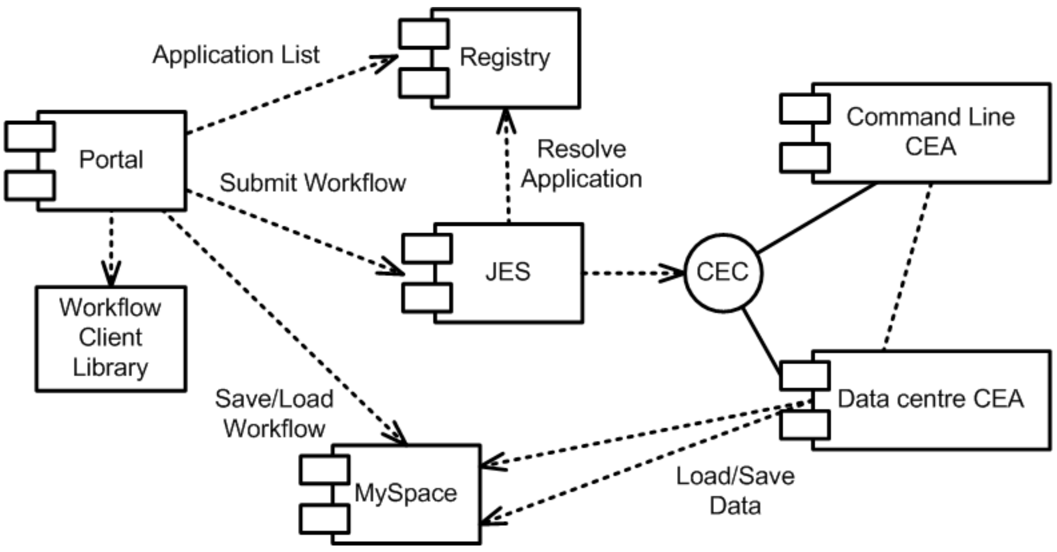


Figure 3.1: Overview of AstroGrid architecture

The AstroGrid architecture is based around the construction and execution of workflows. The architecture distinguishes data-processing work, including archive queries, from other operations such as browsing directories of resources or administering the system. Data processing is always achieved through workflows; the other operations are done interactively, through the web portal. A scientist using the AstroGrid system must construct a workflow into a scientifically meaningful experiment. Workflows are set up graphically through a web portal and executed asynchronously as batch jobs. Where a desktop scripting language would have calls to local programmes, the AstroGrid workflow engine makes calls to remote web services. Although initially, a scientist must learn a new set of skills in order to compose and execute workflows, the steep learning curve means that workflows become a source of intellectual capital. The workflows can be reused, refined over time and shared with other scientists in the field.

AstroGrid can be defined as a *job-oriented* system. A job being a running instance of a workflow. AstroGrid is built under the assumption that the virtual observatory will be used for processing large, complex processing jobs, a job-oriented system makes this more inherently more scalable. The current release is AstroGrid 2, which consists of a number of core interacting components, an overview of which is presented in Figure 3.1. Each component will now be discussed in relation to Figure 3.1:

- **Portal:** A user interacts with AstroGrid through a web based portal. Here a user can perform a number of tasks: explore their MySpace directory, browse exposed resources, create and run a workflow, construct queries and monitor currently running jobs.

- **Registry:** Collections of databases in astronomy are diverse in size, content, location and data formats. Tools have been built up over many years, written in different programming languages and executing on different operation systems. The AstroGrid Registry is the first port of call when a user needs to locate a service capable of performing a particular function, e.g. a data archive which contains information on clusters of galaxies. It takes the complications of manually searching for these services and abstracts details which are unnecessary to the user; such as where the service is located, what language it is programmed in etc.

- **Job Execution Service (JES):** When a workflow begins execution it is treated as a job. A workflow becomes a job by submitting it to the Job Execution Service (JES). A job is a specialised workflow document, containing additional run-time information which allows it to execute. JES is AstroGrid's workflow engine and can manage jobs consisting of multiple steps, where individual steps can be run on different computers on a network. JES will then attempt to run all the steps based on the workflow definition. Each step of the workflow is executed asynchronously, as is the entire job. Once the job has finished executing the user is informed and the results are published in the user's MySpace account.

- **MySpace:** This is the virtual file system used by AstroGrid. It gives the illusion of a directory tree of one system, when in fact files may be distributed across many servers. Storage services in MySpace are split into 2 kinds: MySpace *managers* support the distributed directory tree and *filestores* provide the physical data storage. MySpace provides each astronomer with a homespace. As a workflow progresses, and intermediate data is generated, this data is stored in the users MySpace account. When a workflow has finished executing, the overall output is also stored in the user's MySpace.

- **Common Execution Architecture (CEA):** Everything in AstroGrid is exposed to the system through the Common Execution Architecture (CEA). The CEA is essentially a standard interface which describes how to execute a typical Astronomical application within the Virtual Observatory. This allows any data centre or data processing tool to be accessed in exactly the same way. Application writers then have the simple requirement of implementing a standard interface in order to expose their application to the VO infrastructure. The CEA offers a higher layer than that of WSDL, by providing specific semantics for astronomical quantities and extra information which is not supported in WSDL.

Virtual Observatory technologies offer the power of Grid computing in a way that allows astronomers to achieve meaningful science. AstroGrid is in a maturing state of development, with recent workshops [22] aimed at teaching astronomers how to use the Virtual Observatory.

3.2 Scenario 1: Batch Processing

Photometric redshifts use broad-band photometry to measure the redshifts of galaxies. While photometric redshifts have larger uncertainties than spectroscopic redshifts, they are the only way of determining the properties of large samples of galaxies. This workflow makes use of INT (Isaac Newton Telescope) Wide Field Survey [59] archive in Cambridge, to retrieve images around a selected position and determine photometric redshifts from U, g, r, i and Z photometry.

Photometric redshifts are often calculated through two well known tools. The first is HyperZ [13], which calculates the photometric redshift using Spectral Energy Distributions (SED). The second is called ANNz [17] which is a software package for photometric redshift estimation using Artificial Neural Networks. ANNz learns the relation between photometry and redshift from an appropriate training set of galaxies for which the redshift is already known. The batch processing scenario uses both algorithms to compute the photometric redshift, comparing the accuracy of each approach. The only technical requirement of a user is to supply an RA and DEC (coordinates) of interest. The two Sections of the workflow are illustrated in Figures 3.2 and 3.3.

The workflow begins with the user inputting the RA and DEC coordinates (defining a patch of sky of interest) into the system. The *Wide Field Survey Archive (WFS)* is queried for images covering the patch of sky outlined in the RA and DEC coordinates. Images from each of the bands [U, g, r, i, z] are retrieved from the WFS database. The images are saved in the AstroGrid *MySpace* storage facility. Each image from the 5 wavebands [U, g, r, i, z] is then run through the *SExtractor* [12] service. This

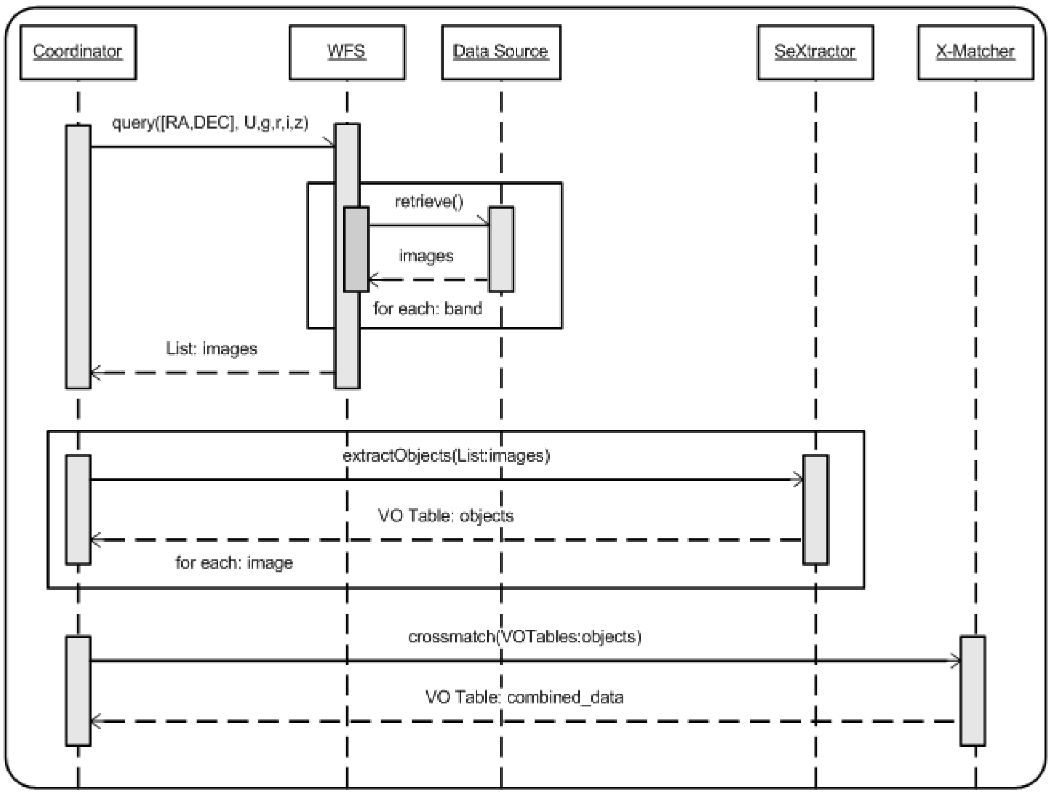


Figure 3.2: Batch processing scenario - obtaining photometric data

application scans the image and uses an algorithm to extract all objects of interest (positions of stars, galaxies etc.) and produces a VO Table for each of the wavebands containing all the data. A *cross matching tool* is used to scan all the images and produce one VO Table containing data about all the objects of interest in the sky, in the five wavebands. This Section of the workflow is detailed in Figure 3.2.

With reference to Figure 3.3 a call is made to a database which contains spectroscopic data covering the same area of sky as the original RA and DEC supplied by the user. An algorithm then needs to compute which galaxies supplied by the spectroscopic database match up with those returned by the merged photometric catalogue (the final stage of Figure 3.2). As the ANNz algorithm use a neural network, it must be trained in order to operate correctly. An appropriate training set and test set is constructed and used to test each of the various configurations of ANNz. Once the neural network is set up, all remaining photometric data can be supplied, resulting in a calculation of photometric redshifts (ANNz: photometric_redshifts). An identical call is made to the HyperZ algorithm, which again computes and returns the calculation of the photometric redshifts (HyperZ: photometric_redshifts). The final output consists of

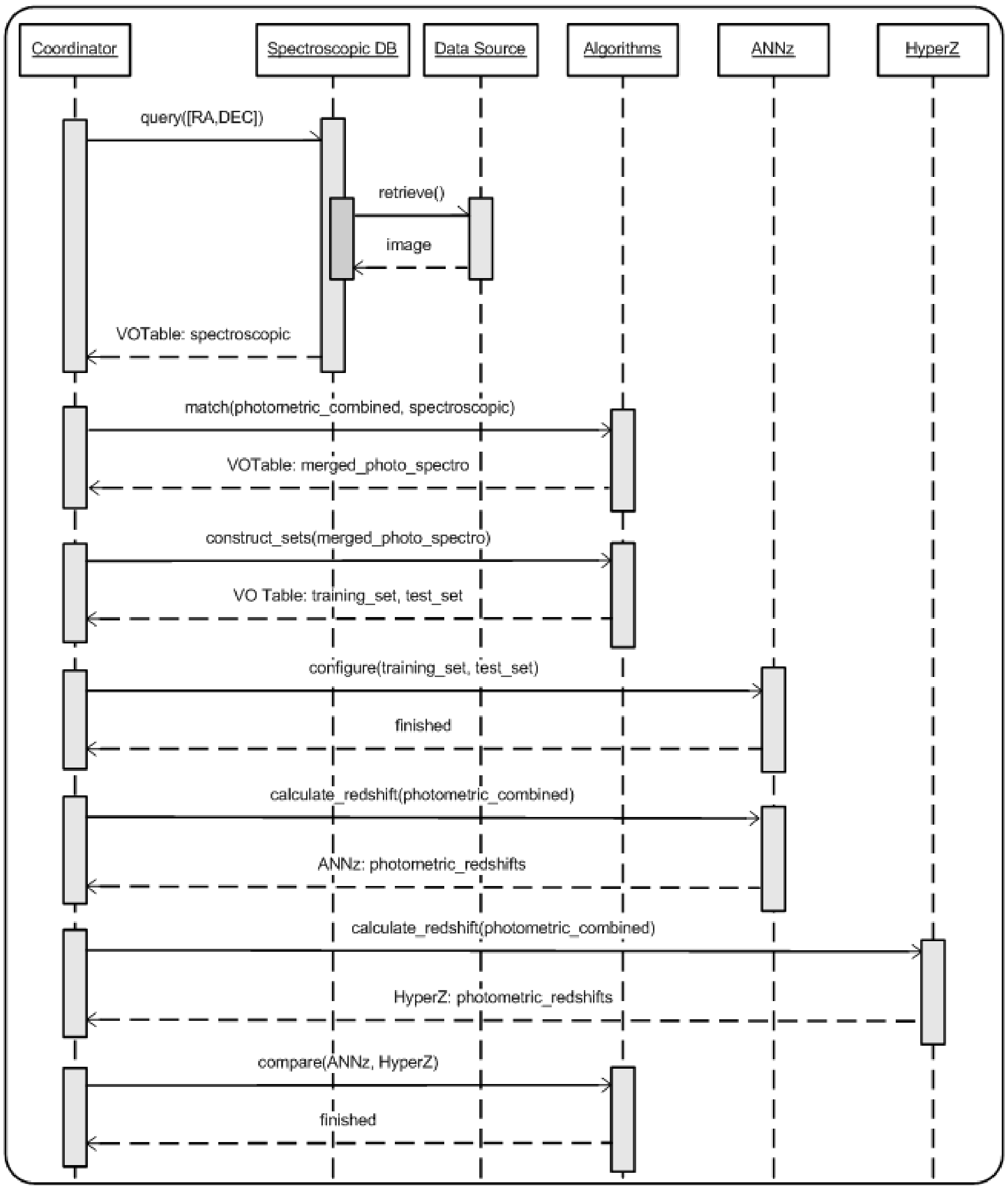

Coordinator
Spectroscopic DB
Data Source
Algorithms
ANNz
HyperZ
query([RA,DEC])
retrieve()
image
VOTable: spectroscopic
match(photometric_combined, spectroscopic)
VOTable: merged_photo_spectro
construct_sets(merged_photo_spectro)
VO Table: training_set, test_set
configure(training_set, test_set)
finished
calculate_redshift(photometric_combined)
ANNz: photometric_redshifts
calculate_redshift(photometric_combined)
HyperZ: photometric_redshifts
compare(ANNz, HyperZ)
finished

Figure 3.3: Batch processing scenario - spectroscopic data, ANNz, HyperZ

multiband files containing the requested position as well as a table containing for each source all the output parameters from SExtrator and HyperZ (or ANNz), including positions, magnitudes, stella classification and photometric redshifts and confidence intervals. A comparison can then be made between the output of the two algorithms.

3.3 Scenario 2: Knowledge Acquisition

If one observes clusters of galaxies with a range of sizes/luminosities, it is often apparent that there is one galaxy which is much brighter than all the others. This galaxy, *called the Brightest Cluster Galaxy* (BCG) is frequently positioned in the centre of the cluster. Statistically, it can be shown that the BCG is something more than just the brightest galaxy in the cluster. Galaxies in clusters follow a fairly general distribution of luminosities, and BCGs are too bright too often to be simply the upper end of that distribution.

There are real outliers, pointing to some different process of formation and/or evolution. The scientific background to this scenario is that there is some evidence for correlations between the properties of the BCG and of the cluster of galaxies in which it resides. This indicates that the cluster is affecting the way that the BCG is formed or has evolved, but scientists do not yet know how this works.

3.3.1 Data Retrieval

An astronomer has a hypothesis about connections between the properties of BCGs and those of their host clusters, and attempts to construct an experiment which will aid the understanding of this subject. The data retrieval Section of the workflow is illustrated in the UML sequence diagram, Figure 3.4. The experiment begins with a query being constructed and sent to the Virtual Observatory (VO), in order to obtain a sample of cluster/BCG pairs which have been well observed in a number of passbands. This is achieved by performing the following operations:

- A query is made to the VO registry web service, in order to obtain a list of VO data sources which are classified as being catalogues containing clusters of galaxies. This yields a list of, say, a dozen cluster catalogues - some based on optical/near-IR observations, some on X-ray observations, and some on submillimetre observations. Examples of real astronomy data sources include: The XMM-Newton Science Archive (X-Ray) [62], the Sloan Digital Sky Survey (SDSS) [60] and the UK Infrared Deep Sky Survey (UKIDSS) [32].

- Each catalogue is referenced individually and from each catalogue the positions of all the clusters are extracted. This is illustrated by the loop between the coordinator and the cluster catalogues in Figure 3.4.
- The VO registry service is referenced again, this time for data sources which are classified as catalogues of optical, near-infrared and radio sources (and which, therefore, might include relevant observations of BCGs). This yields a set of perhaps a further 100 databases.
- At each database the query web service is invoked, in order to extract all the attributes of all sources contained in a search radius of a certain size around the position of each of the clusters (returned by the first service). Illustrated by the loop between the coordinator and the data sources in Figure 3.4.

Once this process is complete these data are deposited in the AstroGrid storage facility, MySpace. Here the data can be accessed for further investigation and analysis by the later stages of the workflow.

3.3.2 Data Analysis

Once the data have been deposited, an analysis routine can begin execution. This analysis routine has to work out which galaxies in the galaxy catalogue data are the BCG's in each of the host clusters and generate a combined set of all the data known about each Cluster/BCG pair. The BCG algorithm must retrieve all the stored data from the MySpace facility, shown as the first part of Figure 3.5.

Let us assume that this procedure yields a set of up to 400 attributes for 10,000 BCG/cluster pairs; not every BCG/cluster pair has a value for each attribute (a measured property which is recorded), but most have values for the great majority of them. This is deemed to be a good working data set. So, for each cluster in the catalogue there is likely to be a number of properties recorded; obvious things like position, brightness, size etc. There will also be another set of attributes recording properties of the sources extracted from some optical or near infrared catalogue.

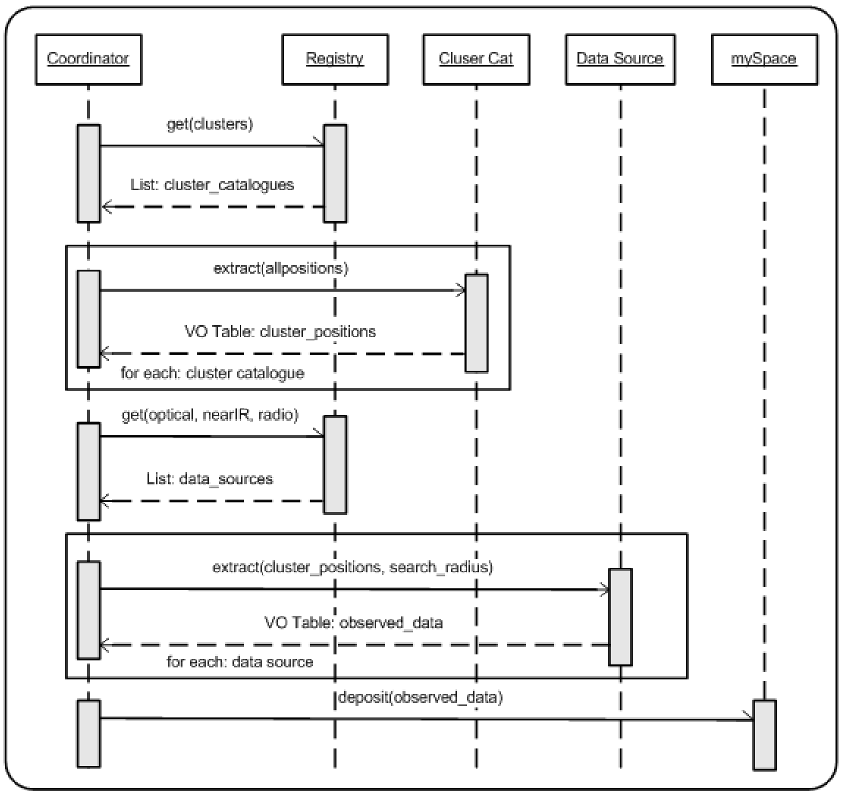


Figure 3.4: Knowledge acquisition scenario - data retrieval

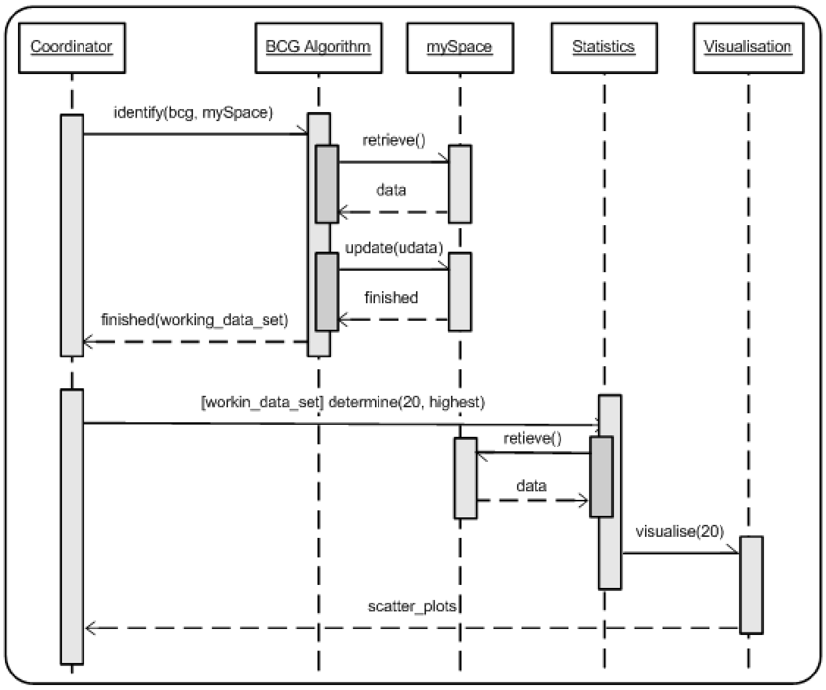


Figure 3.5: Knowledge acquisition scenario - data analysis

The astronomer then runs a statistical algorithm, offered by a web service, which seeks the twenty attributes with the highest information content on the deposited data. The output (attributes returned by the web service) is then fed into a graphics package which generates a grid of scatter plots for pairs of them, arranged in order by the strength of correlation between them. If there are N attributes for M BCG/Cluster pairs then the Grid of Scatter plots represents N*(N-1)/2 plots, each with M points plotted. In other words each attribute is plotted against each other attribute for the set of BCG/Cluster pairs. The visualisation tool allows further investigation into correlations allowing identification of the significant ones.

3.3.3 Data Visualisation

The final Section of the workflow is shown in Figure 3.6. The astronomer must step back and look at the data, the visualisation tool displays a set of scatter plots which are judged as possibly worthy of further investigation. The astronomer must give a sanity check on the statistical correlation tests, since some kinds of correlation are not readily detected by simple summary statistics. The astronomer, after taking a look at the grid of scatter plots reveals that there are very significant correlations between a set of six attributes. So the astronomer launches another visualisation tool, which allows navigation through projections of a multidimensional data space.

The astronomer needs to select a subsample of 200 objects to visualise. A request to the statistical web service is made, quoting the six attributes of interest. The web service uses a statistical algorithm, which makes sure the sample is representative of the full data. These data are then further analysed by the visualisation tool. This reveals three clusters of points, presumably corresponding to distinct populations, which the astronomer defines as three classes.

This classification scheme is then applied to the full set of 10,000 records (i.e. an additional attribute is added to the stored dataset in MySpace, which is the flag for whichever of the classes each BCG/cluster pair belongs to) and statistical tests are run to assess its significance. This is found to be strong, so the astronomer saves the data from this session in MySpace, and moves on to figuring out the astrophysical processes that might lie behind this division into three classes.

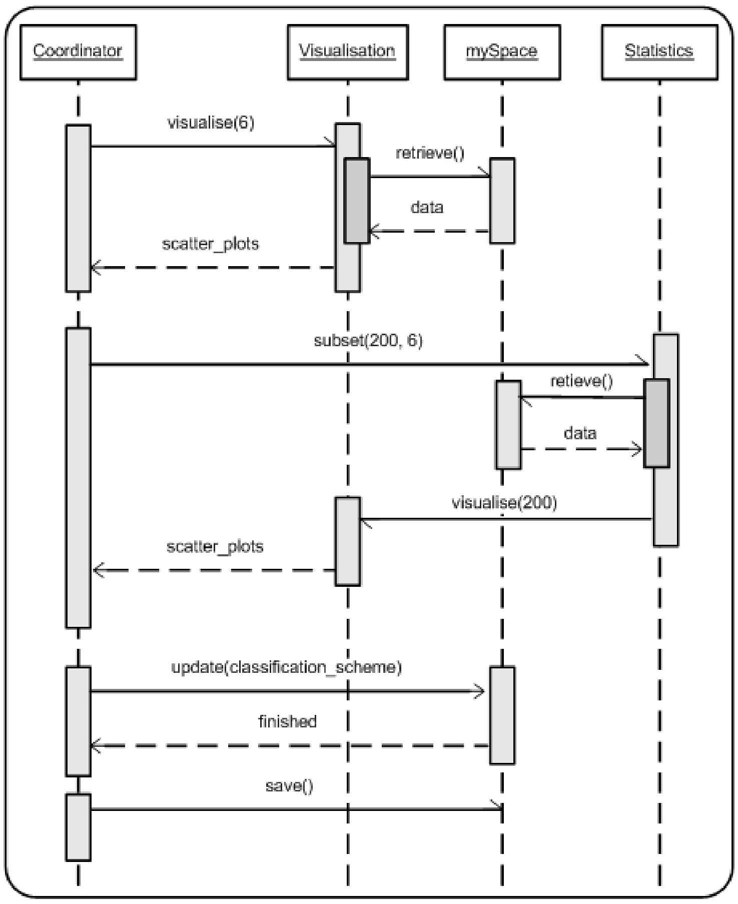


Figure 3.6: Knowledge acquisition scenario - data visualisation

3.4 Requirements Analysis Part I

By analysing the state-of-the-art in scientific workflow systems, in Section 2.2 and presenting a set of concrete workflow scenarios, we are now in the position to address the common requirements of scientific workflow. This Section presents these core requirements in detail.

Scientific and business workflows began from the same common ground. Both communities have overlapping requirements, however they each have their own domain specific requirements, and therefore need consideration separately. Today there is a broad spectrum of Business Process Modelling languages [65], but very few languages which deal with the flexible *knowledge acquisition* and *discovery processes* found in the sciences.

Business workflows place an emphasis on control-flow patterns and events, whereas scientific workflow tends to have an execution model that is dataflow-oriented. A dataflow language models the program as a directed graph of the data flowing between operations. The vast majority of programming languages use the imperative programming model. In imperative programming, the program is modelled as a series of operations (then, or etc.), the data effectively being invisible. Dataflow languages on the other hand treat the data as the main concept behind any program. Programmes expressed in a dataflow language start with an input and illustrate how that input is used and modified. Operations consist of a black box with inputs and outputs. Operations run as soon as all their inputs become valid, as opposed to when the program encounters them, as is the case with imperative programming. A dataflow language is more like a series of workers on a production line, who will complete their assigned task as soon as the materials arrive. Dataflow languages are inherently parallel, as there is no hidden state to keep track of, unlike imperative programmes. A dataflow program will usually be constructed as a big hash table, with uniquely identified inputs as the keys, and pointers to the code as data. When an operation completes, the program scans down the list until it finds the first operation where all of the inputs are currently valid and runs it. When that operation terminates it will typically put data into one or more outports, thereby making another operation valid. Therefore the task of maintaining state is removed from the programmer and given to the language's runtime environment instead, as the only requirement of making a program parallel is to share the list containing the port information.

The dataflow paradigm is used by most of the active scientific workflow projects. As discussed in Section 2.2 myGrid [53] uses a specifically designed simple dataflow language: SCUFL. Kepler [6] is based on the Ptolemy II system, a mature dataflow oriented workflow language, ICENI [41] also has dataflow semantics. All these projects have a selection of components which have input and output ports and it is then up to the user to wire these components together to form an executable program. This is usually achieved through the use of Graphical User Interface. The ten requirements of scientific workflow will now be discussed in detail, with reference to both the active projects and the set of motivating workflow scenarios presented in this Chapter.

3.4.1 Requirement 1: Rapid Prototyping

Scientific work is centred around conducting experiments. A scientific workflow system should mirror a users conventional work patterns by allowing them to apply their methodology over distributed resources. A scientist should be able to work on the process of experiment construction, treating the distributed resources and services as problem solving components, in order to falsify a hypothesis. These problem solving components and the parameters they require need to be continually tweaked by the scientist, until the outcome of the experiment either proves, or disproves this original hypothesis.

A scientific workflow begins as a *research workflow*. The focus here lies on iterative design, steered by a hypothesis. This refinement process can terminate when a suitable combination of workflow components and parameters falsify this original hypothesis. As a result of this incremental design process, scientists require the ability to prototype experiments rapidly. It is therefore essential that the workflow language and scientific workflow system can support this kind of *incremental*, *exploratory* and *prototypical* approach to workflow composition; allowing scientists to quickly test a hypothesis. The myGrid Taverna workbench [43], described in Section 2.2.1 is an example of a scientific workflow system which allows a user to rapidly prototype and execute experiments using a simple dataflow language: SCUFL.

3.4.2 Requirement 2: User Interaction

User interaction is an essential requirement of scientific workflow modelling. There are many occasions where a user will require the ability to choose between different paths of execution, input parameters to a service, modify parameters while the workflow is executing and wire in new workflow components if something fails. A particular type of user interaction known as a *smart re-run* [38] is highlighted in the Kepler system. A smart re-run allows a user to alter parameters while the workflow is still executing. For example a user may (after inspecting the results of the first steps of the workflow) want to alter the parameters and/or components which will affect the following stage. The workflow would not need to be executed from scratch, only the parts which were affected by the parameter changes.

To illustrate the importance of user interaction, we refer again to the knowledge acquisition scenario, presented in Section 3.3. Although the first two stages (illustrated by Figure 3.4 and 3.5) can be executed automatically, there are several steps during its execution where user interaction is required. Once the initial grid of scatter plots has been loaded into the visualisation software, the user must give a sanity check to the correlations. At this point the workflow system will pause execution, waiting for input from the user. It is only through user intervention and the scientists expertise that a significant correlation is found between 6 attributes. A subsample of 200 objects are selected along with the 6 attributes of interest, these are, in turn, used as input to a statistical web service. The scientist's knowledge is again required, this time to derive the classification schema which is applied to the full set of 10,000 records.

The knowledge acquisition scenario highlights the need for a workflow language and scientific workflow system to allow flexible, user-driven interaction, specifically demonstrated by the need for the scientist's expertise in order to find the patterns in the attributes, and to derive a classification schema. There are many processes which simply cannot be automated, and often the best solution is to keep the user in the loop.

3.4.3 Requirement 3: Workflow Reuse

Once the processes behind a scientific workflow are properly understood, a *research workflow* can be executed automatically as a *batch workflow*. As a result of the lengthy iterative process of design, workflows become a *valued commodity* and a source of intellectual capital. These workflows can be reused, refined over time, and shared with other scientists in the field. It is not necessary for a user to understand how a batch workflow is constructed internally, it can effectively be treated as a black box of computation which the user can customise by parameterising the workflow specification.

As an example, take the batch processing scenario discussed in Section 3.2. In the AstroGrid architecture, this workflow is presented to the user simply as a unit of computation, that will, given an input, determine photometric redshifts. The user needs to know nothing of the internal computation, merely supply an RA and DEC (coordinates) for the workflow to begin execution. Once the complex iterative process of designing a scientific workflow has finished, other users can take advantage of this research. It is therefore essential that workflows can be parameterised at runtime and are fully reusable at every stage in the design process.

3.4.4 Requirement 4: Fault Tolerant Execution

The Large Synoptic Survey Telescope (LSST) (discussed in more detail in Section 4.1) will generate 36 gigabytes of data every 30 seconds. Over a ten-hour winter night, LSST will collect up to 30 terabytes of data, and eventually archive more than 50 petabytes. Storing these data alone is a difficult problem, but things become far more complex when users attempt to analyse, retrieve and visualise these massive volumes of data. Scientific workflows are therefore often *data, compute and analysis* intensive. With the movement and analysis of such large quantities of data, these complex workflows can take days, or as long as weeks of compute power to finish one iteration of the workflow execution cycle.

Workflows with execution times as long as this need the ability to run with a *detached execution* mode. This means executing in the background of a parallel machine or Grid cluster, without the need to stay constantly connected to a client's application. Callback mechanisms when the workflow requires user intervention, or parameter changes need to built into the workflow engine. The myGrid notification service [36] is an implementation of such an idea. The scientist cannot afford for the workflow to fail half way through the execution cycle and have to be rolled back to the start. Therefore reliability and fault tolerance factors are important when considering the design of a language and system to support scientific workflows. Common techniques such as transacting, checkpointing [18] and multiple service options (in case a particular service instance is down) need to be built into the language and tools in order to provide the user with fault tolerant execution.

3.4.5 Requirement 5: Suitable Abstraction

A workflow system, particulary when used in the scientific community, should allow the same information to be shown at various levels of abstraction, depending on who is using the system. A high level of abstraction should be presented to a scientist who knows nothing (or simply doesn't care), about the under-pinnings of service composition. The scientific workflow system should present this type of user with an intuitive Graphical User Interface (GUI) or a simple formal notation. An engineer on the other hand, might be interested in the lower level details of exactly how the workflow is composed. Such as where data archives are located and which exceptions a service throws, if it were to fail. These levels of abstraction should be fluid, many scientists will be happy with the high-level definition most of the time, but will want to drill down into the specific details occasionally, e.g. when unexpected results are obtained. A scientific workflow language and system should therefore, be able to present analytical knowledge discovery workflows for scientists, as easily as presenting low level plumbing workflows for software engineers.

myGrid solves this problem by allowing a workflow to be displayed at varying levels of abstraction. A scientist using Taverna can construct a workflow graphically through the user interface, or load and execute a pre-written workflow (such as the batch processing scenario). Whereas an engineer can tweak the individual services and composition control flow with the dataflow language, Scufl [44]. Varying levels of abstraction are essential for both scientists and engineers to make full use of a system.

3.4.6 Requirement 6: Legacy System Integration

Many scientific applications are considered *legacy applications*. These applications tend to be written in older programming languages, such as Fortran. However, just because they are written in older languages, does not mean that they should be discarded. The reason that they are still in use is that they have a proven track record, are reliable and known to work. From a software engineering perspective, it is far more efficient to integrate these applications into newer systems than rewriting the code from scratch.

With the widespread adoption of service-oriented architectures and web service standards, legacy code can simply be wrapped in a standard interface and exposed as a web service. This web service can then, like any other, be used as a building block for more sophisticated applications; such as scientific workflow systems. SExtractor and HyperZ are two examples of legacy applications used by the batch processing scenario. These applications have evolved with time and have been wrapped up as web services in order to be integrated into AstroGrid's service-oriented architecture.

myGrid [53] allows legacy code to be integrated as processor types, such as WSDL or SOAPLAB. Kepler and ICENI have similar mechanisms which allow code to be automatically wrapped and integrated into the scientific workflow environment.

3.4.7 Requirement 7: Provenance Data

As we have illustrated in Requirement 1, scientific workflows can be hypothesis driven, as a result, more often than not the outcome of the workflow will be unsuccessful [71]. It is therefore essential that the system documents the series of steps a user performed which resulted in the unsuccessful outcome. This information may be crucial to aid the evolution of the workflow, in order to produce a successful outcome.

Scientific workflows must be fully reproducible. In order for a workflow to be reproduced, information must be recorded which indicates: where the data originated, how it was altered, which components fitted together to form the workflow, parameter settings etc. This will allow other scientists to re-conduct the experiment, confirming the results. Output of workflows may be used as a basis for future research, either by the scientists who generated the data, or colleagues in a related field. This methodology is consistent with the usual practise of non-computational labs. A useful feature of a scientific workflow system is the ability to automatically generate provenance logs, which can be inspected by others at a later date. myGrid integrates such a feature and allows provenance documents to be linked together, for on the fly annotation [35]. Provenance data also aids users of batch workflows, allowing users to inspect the results of previous experiments, in order to steer their own.

3.4.8 Requirement 8: Smart Component Choice

With pervasive service access, there will inherently be multiple groups of services which perform the same functionality. For example in the batch processing scenario there are two implementations of web services which perform the redshift calculation: ANNz and HyperZ. However some services are inherently more reliable than others, as they make use of better algorithms, have less down time, lower latency etc. It is also true, that certain combinations of services will work together more effectively than others, as there will be less data transformations in between or they are physically located nearer one another.

A scientific workflow system should therefore, ideally assist the user in selecting components. Firstly at design time by suggesting components which are known to work well together based in historical data but also at runtime, based on the current loads of services etc. For this to work, performance data needs to be recorded, services need to be semantically marked up, and brokers [37], [51] need to offer services which closest fit the user's needs.

3.4.9 Requirement 9: Semantic Mark-Up

The Semantic Web, as briefly discussed in Section 2.1.3 allows data to be wrapped in an additional semantic layer. This semantic layer provides metadata (data about data), by giving information well defined meaning, so machines can then begin to reason about them.

Semantic web services allow the properties and capabilities of a web service to be described, using a markup language such as OWL-S [39]. A scientist who is new to the system will not necessarily know which services to use for a particular experiment. By semantically marking up web services, applications can suggest a selection of services based on a user's needs, as discussed in Requirement 8. This helps to remove the often complex and lengthy process of service discovery. Semantic techniques are a useful addition to a scientific workflow system. By marking up data, web services and workflow components, everything is inherently more reusable and easier to discover. The myGrid project makes extensive use of semantic markup techniques in its Taverna workbench.

3.4.10 Requirement 10: Data Presentation

Data presentation is often overlooked when designing scientific workflow systems, pushed aside as a trivial task that can be addressed later in the systems evolution. However as a scientist may know nothing of how a workflow system operates, it is essential to present him/her with an intuitive user interface, also discussed in requirement 5.

Web services require data to be formatted correctly, using different data types and structures. It may therefore, be necessary to pass the output of one service into a filter service, which reformats the data so that they can be passed into the next service. myGrid offers a number of shim services [53], which can automatically perform this task for a user. Although this is essential for the correct execution of the workflow, it may not be necessary to inform the user that this process is even taking place. These tasks can, depending on who the user is (scientist, engineer etc.) be hidden away.

Multiple data types may be used throughout the duration of the workflow, and it is therefore essential that different types of data are presented correctly to the user. Ideally, the underlying scientific workflow system should choose the most appropriate way to display these data. Tools such as graph plotters and visualisation software should be built into the system, and the same data should be able to be displayed to the user in a number of different ways. This requirement is illustrated in the knowledge acquisition scenario. Here visualisation of the data needs to take place in order for the scientist to make an attribute selection, and proceed with the workflow. Kepler and myGrid offer workbench-like facilities specifically for this task.

3.5 Chapter Conclusions

This Chapter has introduced the domain of Virtual Observatory technology, specifically the UK e-Science project AstroGrid. By working closely with this project we have derived and helped to shape a set of workflow scenarios. These scenarios act as a motivating factor behind this research and demonstrate the complex coordination behaviour required of scientific workflow. By analysing the state-of-the-art scientific workflow systems (discussed in 2.2) and motivating scenarios, this Chapter has demonstrated (by example) how scientific workflow has an overlapping set of requirements with traditional workflow modelling, but also has an extra set of requirements and therefore needs consideration separately. As a result of this process we propose a set of core requirements of scientific workflow.

The following Chapter presents a further workflow scenario, taken from another Virtual Observatory project: LSST. This scenario acts as a counterexample of coordination which is difficult or impossible to achieve by the existing service composition techniques. This scenario requires a fundamentally different approach to service composition, a theme which will be explored throughout the remainder of this thesis.

Chapter 4

A Counterexample

This Chapter presents a further workflow scenario which has been jointly derived with the Large Synoptic Survey Telescope (LSST) project, centring around runtime orbit and object classification. The scenario acts as a pivotal point in this thesis and serves as a counterexample to coordination which is difficult or impossible to enact by existing service composition techniques. This counterexample goes beyond the set of requirements which the knowledge acquisition and batch processing scenarios, requiring a fundamentally different approach to the service composition problem.

Section 4.1 introduces the LSST project (Virtual Observatory technology) and discusses the new field of time-domain astronomy and the potential impacts this will have on the astronomy community. Sections 4.1.1 and 4.1.2 outline in detail the counterexample scenario, while Section 4.1.3 discusses an extension to the scenario based on contract negotiation. Section 4.1.4 highlights the extra set of requirements necessary to solve the counterexample scenario presented by this Chapter. These requirements combined with those presented in Section 3.4 form the requirements analysis and motivation for the remainder of this thesis. Conclusions of this Chapter are presented in Section 4.2.

4.1 Large Synoptic Survey Telescope (LSST)

Observations of change in the universe are difficult to obtain. Most change in the universe is so slow that it can never be directly observed, taking place over millions of years; much like the evolutionary processes taking place on Earth. However many of the most remarkable astronomical events occur on human, and even daily, time-scales; these changes have proven the most difficult to observe. Current observatories are able to look very deeply at very small parts of the sky. This small field of view means that any one observation is not likely to catch a transient event in the act, as the observatories are always looking somewhere else. A small field of view means that an impractically large number of separate observations are required to map the entire night sky. Observational facilities are also in great demand, astronomers must apply for scarce telescope time, with the assignment of only a few nights per year to each astronomer. This means that with the lack of continuous observatory access and a global view, astronomers are almost certainly missing out on what is going on in the universe.

The Large Synoptic Survey Telescope (LSST) [56] has been proposed to address many of these difficulties and open up ‘time-domain astronomy’. This ground-based 8.4-metre, 10 square-degree-field telescope will provide digital imaging of faint astronomical objects across the entire sky, night after night. The unique property of LSST is that it is able map the entire night sky very quickly. LSST will tile the sky repeatedly with overlapping images of approximately 10 square degrees. It will be able to tile the entire visible night sky in a matter in 3 days. Current estimates indicate that LSST will generate 36 gigabytes (GB) of data every 30 seconds and over a 10 hour winter night, will collect up to 30 terabytes.

LSST is broadly interested in two categories of objects, the first of these are known as *variable objects*. Variable objects, are as their name suggests objects which vary over time. Examples of variable stars are *periodic variables* (e.g. they oscillate in size and hence brightness) and *aperiodic variables* (i.e. something dramatic happens every now and then which changes their brightness - such as mass transfer between stars). Occasionally, the brightness of a galaxy changes significantly, but that only really happens if a star in it goes *supernova* - in which case it can shine (briefly) as brightly as all the rest of the stars in the galaxy put together.

The second category of objects that LSST is interested are *moving objects*. Moving objects broadly fall into two classes: *stars* and *bodies in the solar system*. The stars move across the sky relative to us because our Milky Way galaxy is rotating and the Sun, and all the other stars, are in orbit around the Milky Way’s centre of mass. To a good approximation, the speed with which a star is seen to move across the sky depends on how close it is to us, so that the only stars which move rapidly (i.e. an appreciable movement on a timescale of a few years) are the ones which are very close to us. Most stars can only be seen moving over timescales of many years; even decades.

That leaves the solar system objects, which are the prime concern of LSST. There are a number of classes of solar system object. Traditionally they have been divided into planets, asteroids and comets, but in recent years it has been realised that the boundaries between all three classes are somewhat vague. Two types of solar system objects are of particular interest for LSST. The first of these are the *trans-neptunian objects* (TNOs), which are objects whose orbits around the Sun have a larger radius than that of Neptune, so that they inhabit the outer regions of the solar system. Within the general category of TNOs there is one class of object, called *Kuiper Belt Objects* (KBOs) which are of particular interest. The Kuiper Belt is a region of the outer solar system (in the sense of a range of orbital radii) in which a lot of objects are found, so it is dynamically stable. What's particularly interesting about KBOs is that they've been sitting in the outer reaches of the solar system for several billion years (or, at least, the material in them has - maybe individual objects aren't so long lived) so they are good probes of the early history of the solar system, which makes them of great interest to people who study the formation of planets and the early evolution of the solar system. The second type of particularly interesting solar system objects are the *Near-Earth Objects* (NEOs). This is, as its name suggests, a catch-all term for anything that passes close to the Earth, and, of course, these are of particular interest because they could actually hit the Earth.

4.1.1 Scenario 3: Runtime Coordination - Automated Stage

The runtime coordination scenario is taken from the LSST science use cases, a motivating factor behind the development of the LSST program. The data reduction and analysis in LSST will be done in a way unlike that of most observing programmes. The data from each image will be analysed and new sources detected before the exposure for the next tile is ready. This means that if anything unusual is detected, normal observation can be interrupted, in order to follow up any new or rapidly varying events. Other observing resources can also be notified instantly, providing a different perspective on the event. As data are collected, they will be added to all the data previously detected from the same location of sky to create a very deep *master image*. LSST will also build up a database of all known KNOs and NEOs and other moving objects. Each

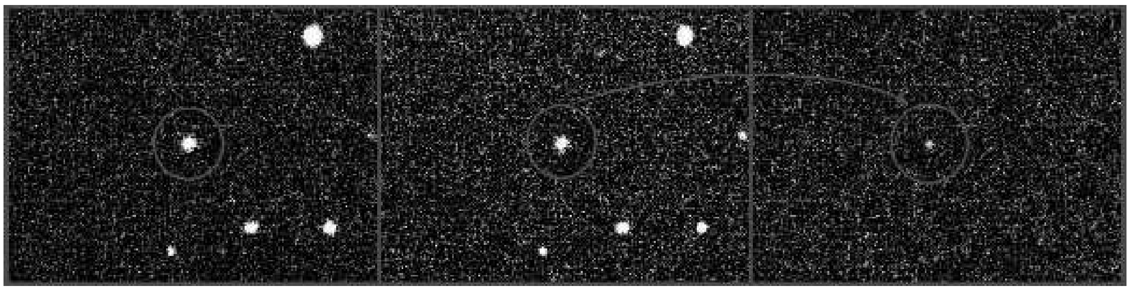


Figure 4.1: An example of a subtracted image

observation will help to improve the accuracy of the data held.

Every time a new image of the sky is obtained, the master image will be subtracted from it. The result is an image which only contains the difference between the sky at that time and its average state; in other words a picture of what has changed, this image is known as the *subtracted image*. Figure 4.1 illustrates two images of a cluster of galaxies, taken three weeks apart, the far right plate is the subtracted image, revealing that a supernova has exploded in one of the galaxies. This subtracted image is then processed by a cluster of computers with the following steps:

- **Find known objects:** This activity runs on the subtracted image. The first task is to compute which objects are expected to appear in the subtracted image, given the area of sky, time of day, and the current state of knowledge of known orbits. This involves making a query to an *orbit catalogue*, which contains data about the orbits of all currently known objects. The results from this query (expected detections) are then cross matched with the sources in the subtracted image. The result is the *unmatched-source catalogue,* this catalogue only contains sources which can't be matched with a previously known object (i.e. the things which may be new discoveries of moving objects). The remainder of the workflow involves attempting to construct an orbit for these newly detected moving objects.

- **Find Tracklets:** The newly created *unmatched-source catalogue* is then used to compute pairs of detections separated by short time intervals, called *tracklets*. A tracklet is an observable, short section of orbit. In order to create pairs of detections, all objects in the *unmatched-source catalogue* are queried against the orbit catalogue, in an attempt to obtain data about these objects from earlier observations. If earlier observations and current observations can be linked to form tracklets, they are stored in the *tracklet catalogue*.

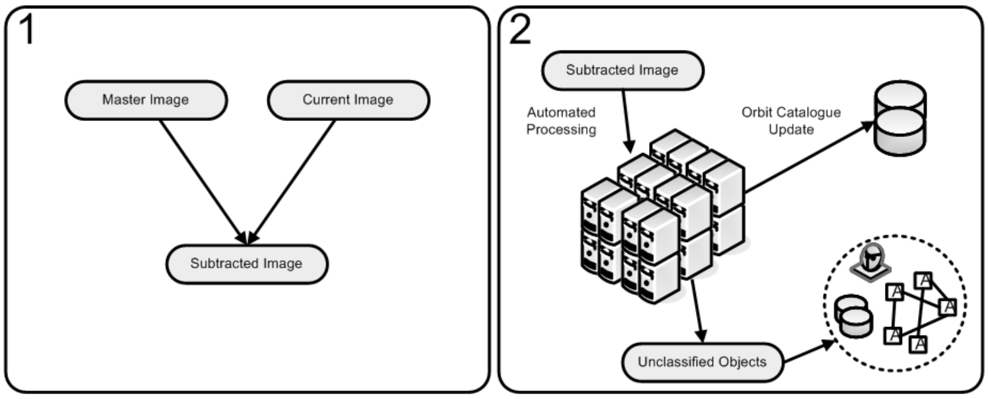


Figure 4.2: Runtime coordination scenario - automated processing

- **Link Tracklets:** The newly created *tracklet catalogue* is then used in an attempt to link these tracklets over a larger time window. This is achieved by again querying the *orbit catalogue*, this time looking for observations of the same objects even further back in time. If matches can be found and the tracklets can be extrapolated out into longer sections of orbit, then they are added as new orbits to the *orbit catalogue*.

- **Orbit maintenance:** The orbit catalogue is then updated, in the light of re-detections of known objects. Each rediscovery provides vital information which helps to constrain the set of known orbits further, resulting in more accurate orbit predictions.

- **Generate alerts:** The final stage of the workflow attempts to classify all entries in the *orbit catalogue* (i.e all the objects which now have known orbits). If any Near Earth Objects are detected to be passing close to the earth, alerts are generated to astronomers so that follow up observations can be scheduled.

After the initial processing stage of the subtracted image (illustrated by Figure 4.2), there will be some data which is left over, the *unclassified objects*. This data includes objects and orbits which can't be classified by the processing software. As LSST is a first attempt at time-domain astronomy, it is likely to discover not only existing types of objects, with many previous, well recorded observations, but also, many new species of objects. If a new species of object were to be discovered, then the automated classification software is almost certain to miss it and the object would end up with the unclassified objects set. This is because no previous data exists about this possible new species of object, so no comparisons can be made to earlier observations.

This leaves us with the question about how to classify the objects in the unclassified objects set. As the content of the data cannot be determined in advance, if a possible new species of object were to arise, it is difficult to ascertain whether it is in fact a new species of object, or simply some kind of observatory equipment failure. Typically, most of these objects will be junk, but this may only be revealed on the basis of comparison with other detections made from the same night.

4.1.2 Scenario 3: Runtime Coordination - Unknown Stage

It is intended that groups of specialised software components take over where the subtracted image processing left off, attempting to classify whatever data is left over from the automated processing stage. The software components are initially set up with a certain amount of knowledge about properties of the data, and a number of statistical tests to perform. They require the ability to cooperate and coordinate with one another, hence they are also set up with some rules about when and how to share information. However, they must be able to react to the constantly changing, dynamic environment which they are operating in. Engineers can focus on developing individual, intelligent software components which are specialised in their own right. For example certain components will have expertise on pixel failures on the camera, others contain data and a hypothesis about a certain kind of unclassified object. Figure 4.3 shows an overview of the example scenario. Observatories are defined within the dotted circle, inside each observatory is a certain amount of local data (illustrated by databases), and a group of software components (illustrated by the square). Web services are shown as rounded rectangles. Communication is shown by arrowed solid lines, web service invocations are shown as single arrowed dotted lines.

An example interaction between a group of distributed observatories could be viewed as the following. Software components at observatory A are attempting to classify objects from the unclassified data set, one of these components has located an item which cannot be classified locally. This anomaly appears on several plates of the sky on the subtracted image, so it wasn't present on the master image. The object and orbit classification algorithms cannot identify the anomaly, so it could potentially be a new species of object, or some kind of equipment failure. The observatory has exhausted the possibility of solving the problem locally and needs to compare similar observations made on the same night with distributed observatories, databases and repositories.

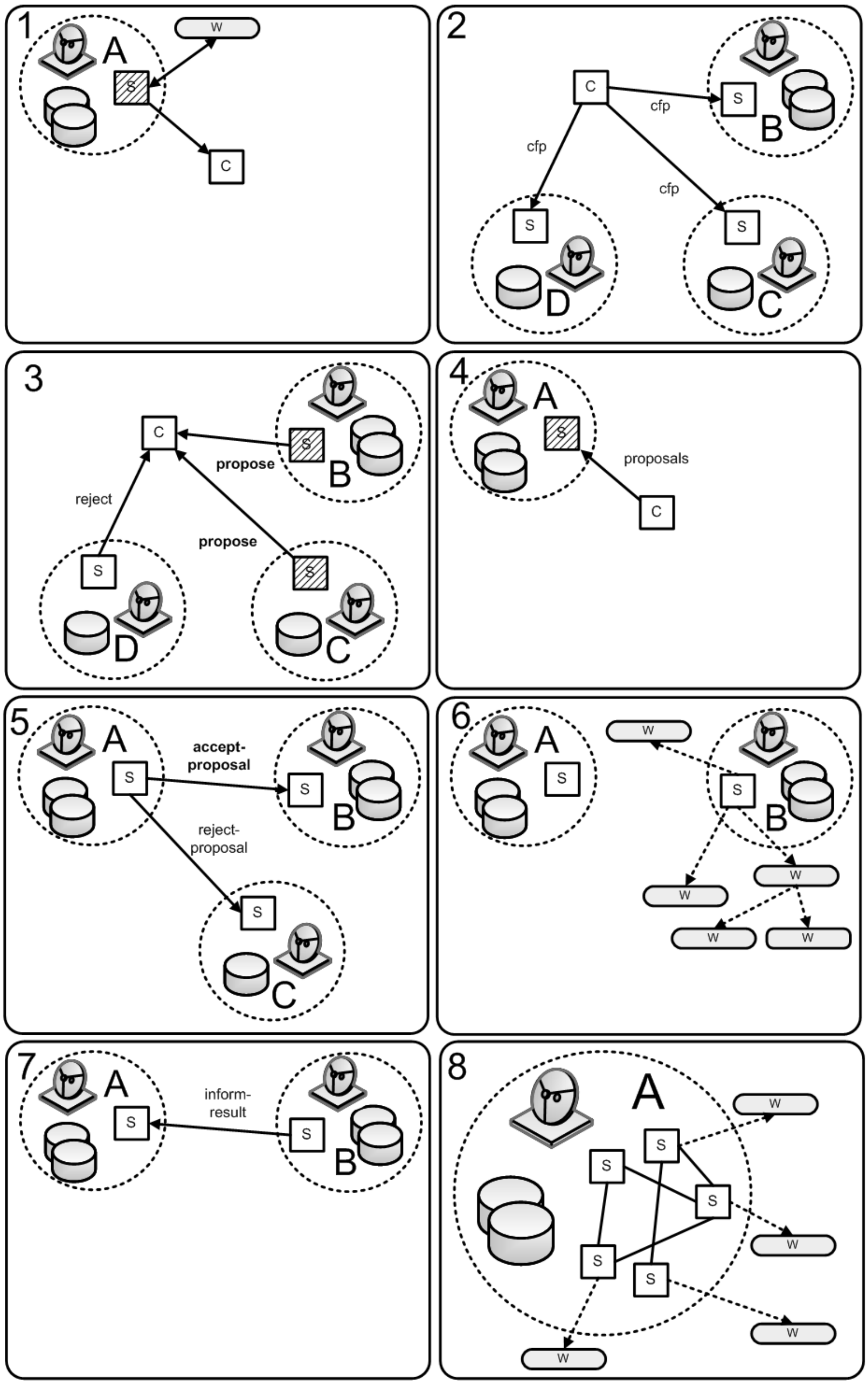


Figure 4.3: Runtime coordination scenario - overview

It wants to ask a question equivalent to: *'has anybody else found anything strange in this particular area of sky, at time t, which could solve this possible anomaly?'*.

In order to discover which observatories can offer the required data, the Contract Net protocol [50] is executed over a group of observatories known to have possible data about the area of sky we are interested in, at time t. This is illustrated by steps 1 to 4 of Figure 4.3. A Contract Net service (on behalf of the observatory) issues a call for participation over the set of possible observatories. The call for participation contains a proposal, defining the terms of agreement. Each observatory then reaches some form of conclusion about participation (based on current work loads, data availability etc.), issuing either an *accept* or *reject* message to the proposal. The set of observatories who returned accept (in this case observatories B and C) are returned to observatory A, who locally decides (based on some internal local knowledge and runtime conditions) which observatory to obtain the data from in order to make forward progress with the classification and workflow. Step 5 of Figure 4.3 shows an *accept-proposal* message being issued to the selected observatory (in this case B) and the remaining observatories are issued a *reject-proposal* message. It is then up to the observatory B to locally retrieve and process the data in accordance to the agreed Contract Net proposal (step 6 of Figure 4.3), this will involve negotiation of resources and a set of external web service calls.

If for some reason the terms of the proposal cannot be met, an *inform-failure* is returned to observatory A. An *inform-failure* will mean that the Contract Net protocol will need to be executed again. Due to changing circumstances, such as network load or scheduling, there is no guarantee which nodes will be available for participation, with this iteration of the contract net protocol, possibly an entirely different set to the original iteration will be available. However, observatory A runs the same process illustrated from steps 1-6 until suitable data is obtained for the workflow. When this occurs an *inform-result* message containing the required data is sent back to observatory A. Once received local software components can use the evidence gathered from the distributed observatories and databases to reach a conclusion regarding the unknown object, reporting anything to human scientists which may require closer inspection. The observatory software then continues to process the remainder of the unclassified data, following the same process again if an object cannot be classified locally.

4.1.3 Scenario 3 Extension: Contract Negotiation and Scheduling

This Section details an extension to the runtime coordination scenario, which deals with contract negotiation and automated observation scheduling in order to follow up any potentially interesting objects. Figure 4.4 illustrates the extended scenario, it picks up where the scenario described in Section 4.1.2 leaves off.

Based on the combined opinion (from the data retrieved from observatory B) the software at observatory A detects that the object in question is not in fact a fault but a possible new species of object, previously undetected. It autonomously makes a decision to schedule an extra observation as quickly as possible, in order to gather further evidence of this new species of object. Observatory A generates a new proposal (based on the area of sky to be scanned) and retrieves a list of observatories from a registry which could potentially schedule the observation (step 1 of Figure 4.4). The Contract Net protocol is executed once again across the group of distributed observatories deemed to be suitable from the registry lookup (step 2 of Figure 4.4). This time instead of simply replying with *accept* or *reject* messages like the previous scenario, there is a further option to propose amendments to the contract. In this case observatories B and D cannot fulfill the terms of the contract and issue a reject message. Observatory C on the other hand can potentially offer the requested service, however due to its current work load it sends back a propose message with a list of amendments attached (step 3 of Figure 4.4). This proposal along with the amendments suggested are sent back to observatory A (step 4 of Figure 4.4)

Observatory A is unhappy with the restrictions placed on the original contract and itself amends the contract again. An iterative process of negotiation takes place between the two distributed observatories (step 5 of Figure 4.4), each making amendments until a draft of the contract is agreed by both parties. If the contract is agreed observatory C places the observation request on the queuing system for the telescope hardware. The place in the queue has been negotiated between observatories A and C, depending on the urgency of the update needed (step 6 of Figure 4.4). Once the telescope has performed the observation the data is sent back to observatory A (step 7 of Figure 4.4). This can then either by used for further processing to confirm or deny a local hypothesis or sent to a group of scientists who have requested notification updates of any potential new species of objects (step 8 of Figure 4.4).

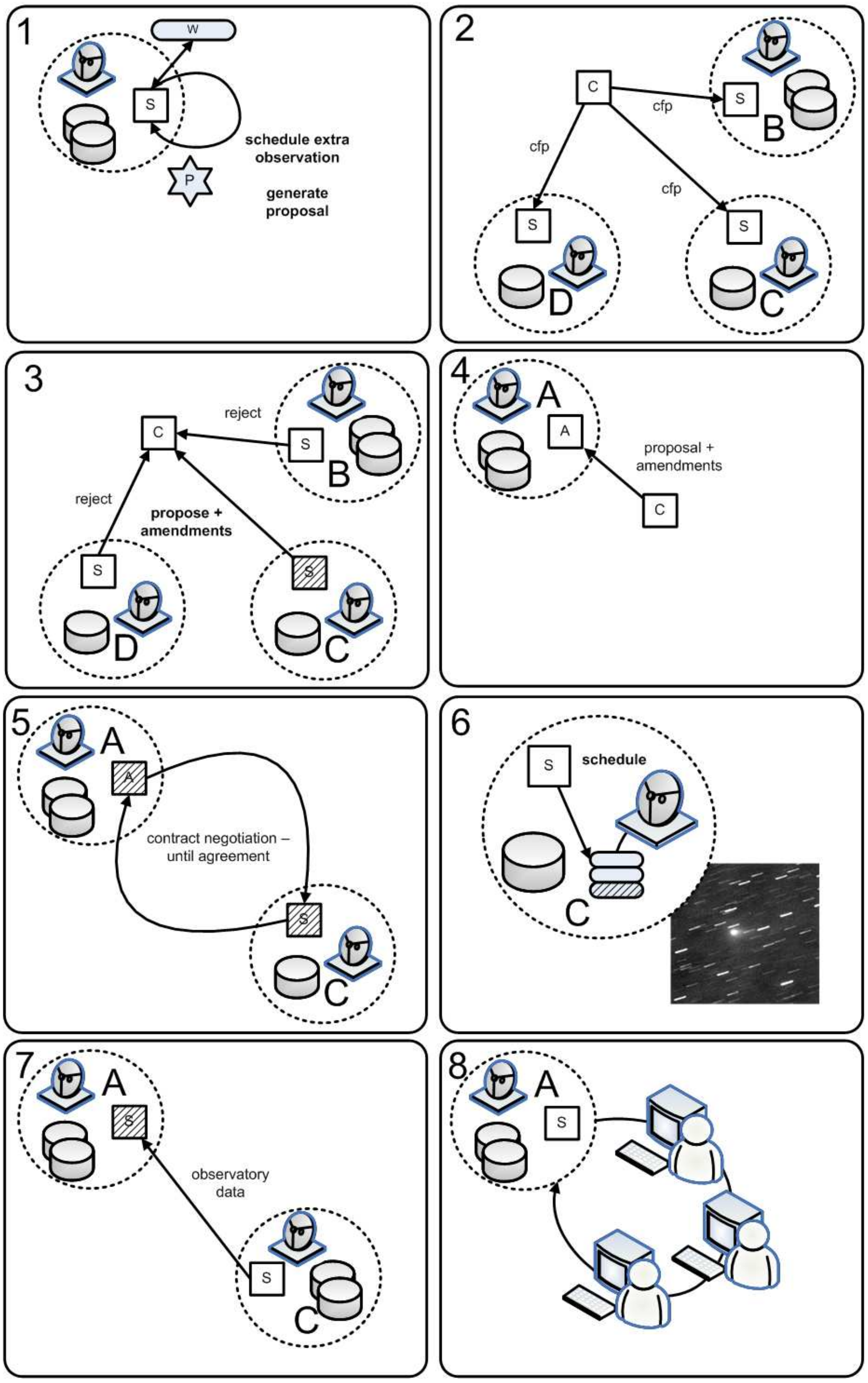


Figure 4.4: Runtime coordination scenario - contract negotiation/scheduling

4.1.4 Requirements Analysis Part II and Problem Statement

Current service composition techniques allow *statically defined*, *pre-designed/pre-planned* workflows to be enacted by a *centralised workflow engine*. Workflows like the knowledge acquisition and batch processing scenarios discussed in Sections 3.3 and 3.2. The scenario discussed by this Chapter acts as a *counterexample* to coordination which is difficult or impossible to achieve by current service composition techniques.

The systems which attempt to classify this data will need to exhibit complex coordination behaviour and go beyond the requirements defined in Section 3.4. These properties will now be discussed in turn:

- **Distributed data:** Data will inherently be distributed over a number of observatory nodes. Systems will need to *collaborate* with these nodes, in order to retrieve the necessary data to make forward progress with the workflow.

- **Scarce resources:** Resources required as part of the workflow (such as sky data and observation time) are scarce. *Negotiation* will need take place if several observatories are either offering the same information/services or bidding on the same resource. This point is illustrated by a possible contract negotiation process, detailed in Section 4.1.3.

- **Data volume:** Due to the quantities of data involved with the LSST project (discussed in Section 4.1), the process of analysing the unclassified objects will need to be performed *autonomously* by *intelligent* software entities. It is not feasible to expect scientists to process this data by hand and human scientists should only be included in the loop if something particularly interesting has been detected, requiring the skills of a specialist scientist. For example when observatory A has gathered evidence (by scheduling observation time) for a possible new species of object, part 8 of Figure 4.4.

- **Runtime coordination:** As there is no way to tell how much, or what type of data will be found in the unclassified objects set, a traditional static workflow, which has been put together at design time, or pre-planned will not offer the flexibility required of this constantly changing environment. The software entities which enact the workflow will need to compose sections of the workflow (which services to call, databases to invoke) *dynamically* at *runtime* to cope with this *uncertainty*. For example the components willing to take part in the interaction changes each time the Contract Net protocol is executed.

- **Pro-activity:** Software components require the ability to be *proactive* in nature, searching and composing solutions to problems they encounter.
- **Partial knowledge:** Observatories may not want to share their data directly, due to privacy issues, funding bodies etc. For this reason it may not be possible to move all the data into one place at the same time (like a traditional centralised workflow engine) in order to run the necessary processing algorithms. Software entities may therefore only have partial knowledge of their environment, this suggests adopting a purely *decentralised, peer-to-peer architecture*.

4.2 Chapter Conclusions

The runtime coordination scenario presented by this Chapter acts as a counterexample to coordination which is difficult or impossible to achieve by existing service composition techniques. In order to achieve the added flexibility required of this counterexample and the requirements discussed in Section 3.4 this thesis views the service composition problem in a fundamentally different way. An agent-based architecture is proposed, allowing active, autonomous agents to consume the passive service-oriented architectures found in Internet and Grid systems.

Specifically we propose modelling the processes found in scientific workflow with the flexible coordination technique of interaction protocols (discussed in Section 2.3.4) from the field of multiagent communication. This has allowed the typical features and requirements of a scientific workflow to be understood in terms of pure coordination and executed in an agent-based, decentralised, peer-to-peer architecture.

For completeness it is important to mention that there are a minimal number of workflow projects based on multiagent/peer-to-peer architectures: Little-Jil [72], PeCo [58], SwinDeW [76], Pockets of Flexibility [48] and WASA2 [67]. Although this thesis recognises the contribution of these projects many are still in their infancy and are merely suggested approaches. None of the approaches are founded on interaction protocols or deal specifically with agent-based service composition for scientific workflow scenarios, like those discussed throughout this thesis.

In the following Chapter the MultiAgent Service Composition (MASC) language is introduced, this agent-based service composition language is specifically designed for modelling complex workflow scenarios like the runtime coordination scenario detailed by this Chapter. The core aims will be be discussed along with a break down of the language syntax and examples of use where appropriate.

Chapter 5

MultiAgent Service Composition (MASC)

This Chapter introduces the formal language: *MultiAgent Service Composition*, otherwise known as *MASC*. The language presents an agent-based solution to the service composition problem and is centred around the concept of interaction protocols. Section 5.1 reiterates the conclusions drawn from previous Chapters and presents the motivations and core aims of the MASC language. Section 5.2 presents in detail the formal syntax of the language, discussing in turn, each construct with an example of use where appropriate. Section 5.3 concludes the Chapter and the full MASC syntax is presented in Section 5.4.

5.1 Service Composition through Interaction Protocols

As demonstrated through our analysis of current scientific workflow systems and motivating workflow scenarios, scientific workflow has an extra set of requirements which go beyond traditional Business Process Modelling. Although the field of scientific workflow is maturing, there are still few languages and systems which deal with the flexible *knowledge acquisition* and *discovery processes* found in the sciences. Through our analysis, this thesis has derived a set of core requirements for scientific workflow.

Section 4.1 highlighted a counterexample which demonstrated that statically defined, pre-designed/pre-planned workflows were too brittle for the flexible, dynamic composition required of the runtime coordination scenario. An extra set of requirements

Traditional Requirements	**Extra Requirements**
R1: Rapid Prototyping	**E1:** Dynamic, Runtime Composition
R2: User Interaction	**E2:** Peer-to-peer Architecture
R3: Workflow Reuse	**E3**: Peers are Autonomous
R4: Fault Tolerant Execution	**E4**: Peers capable of Reasoning
R5: Levels of Abstraction	**E5:** Proactive, Reactive Peers
R6: Legacy Systems Support	
R7: Provenance data	
R8: Smart component choice	
R9: Semantic mark-up	
R10: Data presentation	

Table 5.1: Requirements analysis

(discussed in Section 4.1.4) is required to achieve this complex coordination. This extra set of requirements goes beyond the existing functionality provided by current service composition techniques. This has resulted in a combined set of requirements which together form the requirements analysis for the remainder of this research, to reiterate, an overview of these combined requirements is illustrated by Figure 5.1. The MultiAgent Service Composition (or MASC) language aims to meet these requirements by adopting an agent-based approach to service composition, our approach is founded on the concept of interaction protocols.

5.1.1 Combined Requirements Analysis

Interaction protocols (addressed in more detail in Section 2.3.4) are essentially a collection of conventions which allow agents in an *open system* to interact with one another. The term *open system* means that any agent can take part in the interaction, regardless of their internal implementation details. Interaction protocols define the *rules of engagement* between a group of interacting agents. Such as *if and when* an agent can communicate, and the *order* and *kind of messages* that an agent expects. Interaction protocols sit between the transport layer (defining network specifics, e.g. HTTP/SOAP etc.) and the rational layer (how the agent reacts when it receives certain messages).

Our approach builds on the Electronic Institutions (E.I) framework, which was discussed in Section 2.3.5 and extends an earlier version of the language: Multi Agent Protocols (MAP) [69], [10]. Although Electronic Institutions provide a standard framework for coordinating the interactions of agents in an open multiagent system, there are several problems which prevent it from becoming a truly adaptable standard in the agent community:

- **Centralised control:** All interactions that take place in the EI framework are coordinated through a central agent, known as an *administrative agent*, or *governor*. The administrative agent's job is to enforce the conventions of the institution, and to make sure that agents adhere to the institutions rules and regulations. This is regarded as a bottleneck in the system, because coordination of the agents hinge on the administrative agent functioning correctly, if the administrative agent crashes or performs an incorrect function then all agents taking part in the institution will not behave as expected. One of the key properties of agents (discussed in Section 2.3) over other software entities is *autonomy*, the presence of an administrative agent undermines this key property.

- **Non-determinism:** Protocols controlling the flow of an agent can contain multiple transitions between states within a given scene. However the protocol does not attempt to define when an agent should choose a particular state over another, this is left up to the engineer of an individual agent who must manually assign behaviours to each of the choice points in a protocol. This means that the protocol can not be automatically disseminated to an agent. Human intervention is always required to supply the behaviour at all choice points in the protocol. This can be considered a heavy weight engineering task, especially if the protocol is more complicated than the simple general practitioners scenario discussed in Section 2.3.5.

- **Design time:** The path of an agent through an institution (i.e. the roles an agent will adopt and the scenes in which it will interact) must be determined before the agents are deployed. When a new agent wishes to participate in an institution it must know the internal details of the institution. The current approach is to construct a plan for an agent based on knowledge of a particular institution. This means that the path of an agent needs to be pre-determined. This approach is fine for simple cases, with relatively predictable interactions, however it breaks down in more complicated cases where it is not possible to predetermine the path of an agent through the institution.

- **Static topology:** It is assumed that the structure of the institution remains static. If the definition of the institution changes then the plans for the individual agents will need to be re-synthesised and corrected. This means that even a minor change in the definition of an institution, may mean complicated reworking of all the individual agents. This is highly undesirable and makes the system complicated to deploy and brittle to any changes in the topology.

The MASC language borrows certain concepts from the E.I framework, such as: coordination being defined using interaction protocols, the dividing of these protocols into scenes, roles and agents adopting roles from a protocol. MASC however is an interaction protocol specification language designed to address several of the shortcomings of the E.I. framework and is targeted specifically at service composition, a topic not addressed by the E.I. framework. The MASC language has the following core aims:

- **Uniting agents and services:** The MASC language aims to bridge the gap between the multiagent system and service-oriented architecture paradigms. By applying the principles and well understood practices of agency to the service composition problem. Active, autonomous agents can consume the generally passive service-oriented architectures found in Internet and Grid systems.

- **Peer-to-peer architecture:** Workflows are required to be executed in a decentralised, peer-to-peer architecture, therefore each peer must be able to directly execute the workflow specification. As we have demonstrated most workflow engines are centralised, job-oriented systems, so this shift to a peer-to-peer architecture presents a new set of challenges.

- **Component autonomy:** The language should allow concepts specific to agency to be explored, for example to allow peers a degree of autonomy engineers should be able to integrate specific reasoning models alongside the specification of interaction.

- **Requirements of scientific workflow:** The MASC language aims to meet the combined set of requirements which have been discussed in detail throughout this thesis, supplying the coordination necessary to solve the counterexample workflow scenario presented in Section 4.1. With reference to Figure 5.1 the MASC language directly addresses requirements: R1 - R6, R8 and E1-E5. Discussion of how to achieve requirements: R7, R9 and R10 are presented in Section 8.2, further work.

- **Levels of abstraction:** The language is required to be used at various levels of abstraction, ranging from a scientist: who simply wants to wire together problem solving components in an attempt to nullify a hypothesis (e.g. by taking advantage of a simple dataflow paradigm), to an engineer: who is interested in specific details of service interaction.
- **Framework:** In order to test the ideas presented by this thesis, a framework will be implemented which takes advantage of the latest service-oriented standards. This framework will be made fully open-source.
- **Fit in with existing infrastructure:** As there are several fully developed graphical service composition tools (e.g Taverna [43]), scientists should be able to integrate components expressed in the MASC language into these existing frameworks. For example, adding our novel multiagent/service-oriented approach as a dataflow node in an experiment constructed using Taverna.

5.2 MASC Language Syntax

This Section presents the abstract syntax to the MASC framework, an agent-based approach to the service composition problem. Where appropriate the Backus Naur Form (BNF) notation is used. The language will be discussed bottom up, beginning with the definition of a scene. The notation used is an extended form of BNF, where we have adopted the regular expression symbols * to represent 0 or more, and + to represent 1 or more. Superscripts are used to indicate a list, e.g. $R^{(k)}$ is a list with elements R of size k. Different types of term are represented by prefixing variable names with: $, constants with: ! and identifiers with: %.

5.2.1 Terms, Types, Identifiers and Configuration Pairs

Several elements in the MASC language need to be uniquely identified, this is achieved through the `id` set, consisting of seven elements $\{id_p,\ id_s,\ id_r,\ id_a,\ id_m,\ id_{pin},\ id_{pout}\}$. These identifiers will be referenced throughout the remainder of the syntax, they each represent:

- id_p: Protocol identifier
- id_s: Scene identifier, must be unique within a protocol
- id_r: Role identifier, must be unique within a scene
- id_a: Agent identifier, must be unique within a scene)
- id_m: Method identifier, must be unique within a role
- id_{pin}: Input port identifier, must be unique within a scene
- id_{pout}: Output port identifier, must be unique within a scene

Terms are the objects of manipulation in our language. Terms: ϕ are defined as either a variable: v:τ, a wildcard: _ or a constant: c:τ. Associated with a variable or constant is a type: τ. Types, although not specified in the formal syntax can firstly map to the standard set of JAX-RPC supported types: Boolean, Byte, Double, Float, Integer, Long, Short, String, (Arrays and multidimensional arrays are also supported). Secondly a type may map to the id set, allowing for example agents to store variables where a type is mapped to a unique agent identifier.

A configuration pair: config is a generic $\langle$ name,value $\rangle$ tuple used to parameterise a protocol, role and web service definition along with the mappings of ports to a user, file or web service. Definitions of how configuration pairs are used within different contexts will be explained throughout this Section.

5.2.2 Scenes

Two key concepts in MASC are the division of protocols into *scenes* and the assignment of *roles* to agents. Figure 5.1 formally defines the concepts discussed in this Section. Scenes can be thought of as a bounded space in which a group of agents interact on a single shared task. They allow a large, complex protocol to be divided up into smaller, more manageable chunks. Scenes add a measure of security to a protocol, in that agents which are not relevant to the protocol are excluded from the scene. Scenes place a barrier to execution on the agents, execution of a scene cannot begin until all agents have reference to the protocol and have been instantiated. Formally a scene is comprised of an identifier: id_s, a set of role definitions: {R}, a set of agents: {A}, a set of inports: {inport} and a set of outports: {outport}.

S	::=	`scene`(id_s,{R}, {A}, {inport}, {outport})	(Scene)
A	::=	`agent`(id_a, id_r, $\phi^{(k)}$)	(Agent)
R	::=	$\langle \mathrm{id}_r$, $\mathrm{config}^{(k)}$, {M} $\rangle$	(Role)
M	::=	`method` $\mathrm{id}_m \vert \phi$:m($\phi^{(k)}$) = op	(Method)
inport	::=	`inport`(id_s-id_{pin}, τ, boolean)	(Inport Definition)
outport	::=	`outport`(id_s-id_{pout}, τ)	(Outport Definition)

Figure 5.1: MASC formal scene and role definitions

In order to allow a scene to be treated as a composeable element in our language, the scenes's definition contains a set of inports: {inport} and a set of outports: {outport}. An inport is formally defined by linking a scene name to a input port name: id_s-id_{pin}, specifying which scene the port belongs. A type: τ (discussed in Section 5.2.1) indicates the port type. This specifies the type of data that can be written to the port. The final element of the inport definition is a boolean, indicating whether the port must be written in order to start the execution of the scene. A value of true represents a core port (must be written to) and false a non-core port (execution is not port dependant). For example: `inport(Scene1-in1, xsd:string, true)` represents an inport (named in1) belonging to a scene named: Scene1, the port accepts data of type xsd:string and it is a core port (indicated by the true value in the final parameter).

An outport consists of the same elements as inport without the final boolean value. Ports act as FIFO (First In First Out) queues. Any agent within the scene can consume data from a port using the portread operation and write data to a port using the portwrite operation, this will be discussed in more detail in Section 5.2.3.4.

The concept of a role is central to our definition. Each agent in the set: {A} must adopt an initial role from: {R}. A role determines which parts of the protocol code an agent can execute. Roles allow agents to be grouped together, many agents can share the same role, which means the agents have the same capabilities. A role type allows us to specify a pattern of behaviour which an agent can adopt, this means that we don't have to create a separate protocol for each individual agent. Roles also allow us to specify multicast communication in MASC. For example, we can broadcast messages to all agents who have subscribed to a particular role.

Roles are defined by a unique identifier: id_r, a set of methods: $\{\mathrm{M}\}$ and a list of configuration pairs: $config^{(k)}$. In this instance, configuration pairs are used to represent where the default implementation for an agent role resides, along with the maximum and minimum number of agents that can adopt the role. The behaviour of a role is defined by a set of methods: $\{\mathrm{M}\}$. Methods are constructed from an operation set op, and a set of actions α, more specific details will be discussed in Section 5.2.3 and Section 5.2.4. The final element in the scene definition is a set of agents: $\{\mathrm{A}\}$. An agent is defined by a unique agent name: id_a, and a role identifier: id_r, indicating a role definition residing in $\{\mathrm{R}\}$. If required parameterisation of the agent is possible through the list of input terms: $\phi^{(k)}$.

5.2.3 Action Set

The behaviour of a role is defined by a set of methods: $\{\mathrm{M}\}$, which are each uniquely named: id_m. A method accepts a list of terms as arguments: $\phi^{(k)}$. The initial method is named main by default. Methods are constructed from an operation set: op, which enforce control flow in the agent and a set of actions: α, which (amongst other functions) allow an agent to interact with a reasoning layer. Actions can have side-effects and fail. Failure of actions causes backtracking of the protocol. The action set and operation sets are formally defined through Figure 5.2.3.

Firstly we shall address the action set, which allows agents to: invoke agent reasoning (decision procedure), invoke external web services, create new instances of agents (agent invocation), send and receive messages between agents, multicast messages, interact with a user (user send, user receive), read and write data from a port (port read, port write). Each component in the action set will now be addressed in more detail by the following subsections, graphically the action set is represented by Figure 5.3.

op	::=	α	(Action)
	|	op_1 `then` op_2	(Sequence)
	|	op_1 `or` op_2	(Choice)
	|	op_1 `par` op_2	(Parallel Composition)
	|	`waitfor` op_1 `timeout` op_2	(Iteration)
	|	`invoke` $id_m \| \phi{:}m(\phi^{(k)})$	(Recursion)
α	::=	ε	(No Action)
	|	proc	
	|	`agent`$(id_a \| \phi{:}a,\ id_r \| \phi{:}r,\ \phi^{(k)})$	(Agent Invocation)
	|	$\rho(\phi^{(k)}) \Rightarrow$ `agent`$(id_a \| \phi{:}a,\ id_r \| \phi{:}r)$	(Send)
	|	$\rho(\phi^{(k)}) \Rightarrow$ `multicast`$(id_r \| \phi{:}r)$	(MultiCast)
	|	$\rho(\phi^{(k)}) \Rightarrow$ `user`$(config^{(k)})$	(User Send)
	|	$\rho(\phi^{(k)}) \Leftarrow$ `agent`$(id_a \| \phi{:}a,\ id_r \| \phi{:}r)$	(Receive)
	|	$\rho(\phi^{(k)}) \Leftarrow$ `user`$(config^{(k)})$	(User Receive)
	|	$\phi^{(k)} =$ `portread`$(id_{pin} \| \phi{:}pin)$	(Port Read)
	|	`portwrite`$(id_{pout} \| \phi{:}pout,\ \phi^{(k)})$	(Port Write)
proc	::=	$\neg$ proc | proc $\wedge$ proc | proc $\vee$ proc	
	|	$\phi^{(k)} = \rho(\phi^{(l)})$ `fault` $\phi^{(m)}$	(Decision Procedure)
	|	$\phi^{(k)} =$ `service`$(ws^{+},\ \phi^{(l)})$ `fault` $\phi^{(m)}$	(Web Service Invocation)

Figure 5.2: MASC formal action and operation set definitions

5.2.3.1 Decision Procedures and Web Service Invocations

Procedures (proc) can either be constructed from a decision procedure or a web service invocation. Firstly, decision procedures serve as the connection between the protocol code, describing the coordination and an agents' internal reasoning model. Each agent interacting within the boundaries of a scene references a set of decision procedures, which is implemented as a set of methods and exposed to the agent as a *reasoning web service*. This is graphically illustrated by the inner two circles of Figure 5.4, agents are represented as circles with (A) inside. When an agent needs to make an internal decision, it invokes methods on this web service; for example the logic which decides how much to bid on a particular item, during an auction.

Given a list of input terms: $\phi^{(l)}$, a procedure will invoke the required method on the reasoning web service: ρ, using the terms as input. If required, it will produce a list of

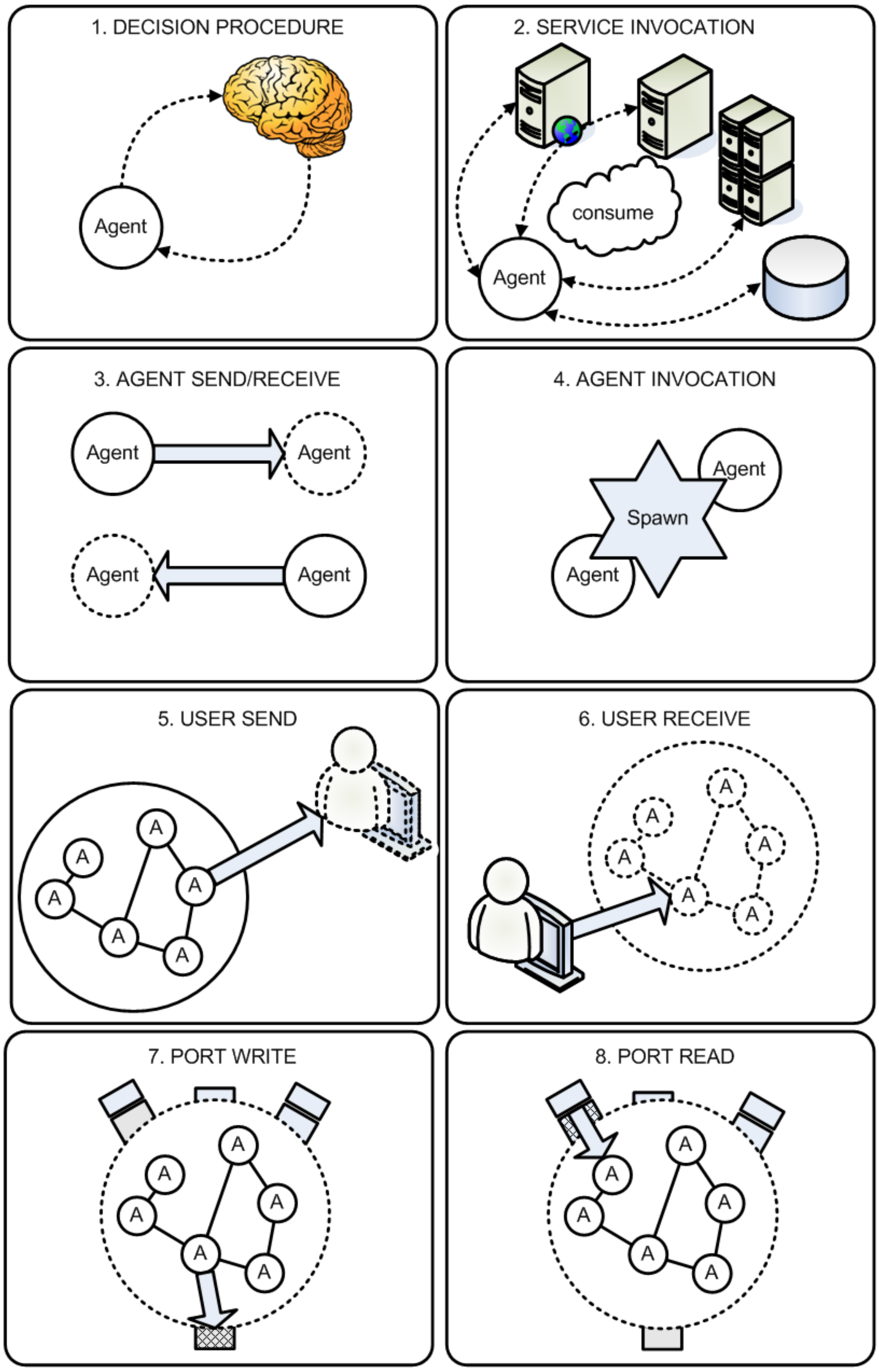


Figure 5.3: Overview of MASC action set

output terms: $\phi^{(k)}$ (results from the procedure) which can be referenced throughout the duration of the agent's execution cycle. A procedure can raise an exception, in which case the exception parameters are bound to the fault terms: $\phi^{(m)}$ and backtracking of the protocol occurs. For example: `$var1 = ProcedureX($var2)` invokes the decision procedure: ProcedureX, using the variable $var2 as input, the output of the invocation is written to the variable $var1.

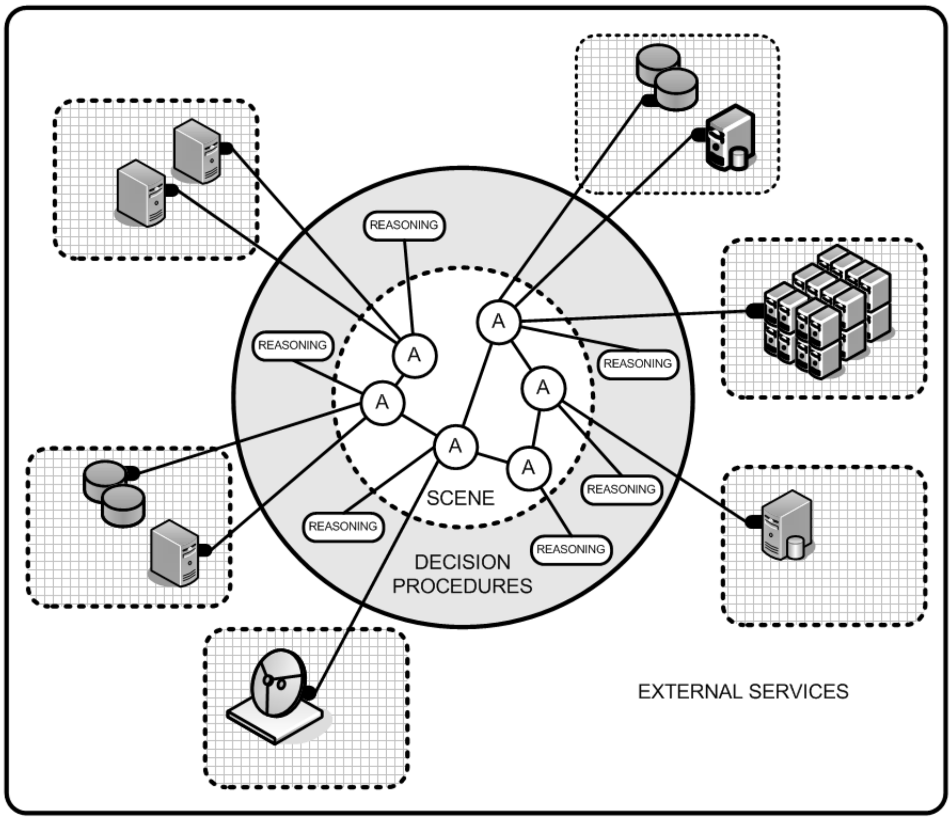


Figure 5.4: Overview of reasoning and external services

This model allows the rules of interaction to be explicitly expressed, while allowing individual agents to subscribe to their own reasoning models, for example Argumentation or the Belief Desires and Intensions (BDI) model [46]. MASC protocols do not sacrifice the self interest and autonomy of individual agents. Although agents follow the protocol as a script, each agent can adopt their own personalised strategy within the protocol. Reasoning web services can be mapped on an individual agent basis (providing personalised behaviour) or by role type (providing generic role behaviour). It is up to the engineer of the agent to provide the implementation (or reference to this implementation) of the decision procedure set which resides in the reasoning web service.

As well as subscribing to a reasoning model, it is essential that agents are able to consume the service-oriented architecture found in Internet and Grid systems, in order to compose multiple services into a scientific workflow. Therefore agents can make direct web service invocations from within the protocol code, illustrated by the outer Section of Figure 5.4. Direct invocations can be made by using the service action. A web service: ws is specified using a list of configuration pairs: $def(config^{(k)})$. An engineer can either hard code the service definitions in at design-time or *they can be resolved at runtime by the agents themselves*. Multiple ws definitions can be used as the first parameter to a service. The first ws definition is always used as the default service to call, the remainder act as backup services, called in the event that a fault arises with the first. This definition(s) along with a list of input parameters: $\phi^{(l)}$ are used to invoke the required external service, binding any output to protocol variables: $\phi^{(k)}$. If exceptions are raised, the parameters are bound to the fault terms: $\phi^{(m)}$. Decision procedures and web services can be chained together using the not: $\neg$ and: $\wedge$ or: $\vee$ operators, allowing more complex behaviour to be defined.

The coordination mechanism defined using the MASC language is entirely external to the web services which are being enacted. The web services themselves need no alteration or knowledge that they are even taking part in the coordination. Therefore no modification of web services needs to take place, and the protocol does not need to be disseminated between the web services themselves. Furthermore, agents add an extra level of abstraction, acting as stubs or proxies to the web services which are taking part in the coordination. This means that the agents can use their rational layer (through decision procedure invocations) to make decisions at run-time, when the web service coordination is actually taking place. Decisions can be taken for example: which services to call, what to do if a particular service is down, how to react if an expected message is not received etc.

5.2.3.2 Sending and Receiving

Interaction between agents is performed by the exchange of messages, defined as performatives ρ, ie. message types. The most commonly used performatives are defined by the FIPA Agent Communication Language (ACL) [1]. Agents can send and receive messages in a number of ways:

- **Specific agent, specific role:** If the first parameter contains an agent id: id_a, or a term representing an agent id: ϕ:a and the second parameter contains a role id: id_r, or term representing a role: ϕ:r. For example: `request($var1)` $\Rightarrow$ `agent(%a1, %role1)` would send the message of performative type: request containing: $var1 to the agent: %a1 who has adopted the role: %role1. This feature is useful for sending messages to specific agents (who are known in advance or looked up at runtime), e.g. to maintain a long-running, consistent dialogue.

- **Specific agent, any role:** If the first parameter contains an agent id: id_a, or a term representing an agent id: ϕ:a and the second parameter contains a wild card: _. For example: `request($var1)` $\Rightarrow$ `agent(%a1, _)` would send the message of performative type: request directly to the agent: %a1.

- **Any agent, specific role:** As there is the possibility that many agents have adopted the same role, a useful feature is the ability to send and receive messages from any agent who has subscribed to a particular role. This is achieved if the first parameter contains a wildcard: _ and the second parameter contains either a role id: id_r, or term representing a role: ϕ:r. For example: `request($var1)` $\Rightarrow$ `agent(_, %role1)` would send the message of performative type: request to any agent who has adopted the role: %role1.

- **Any agent, any role:** If an agent simply wants to send a message regardless of agent id or role id this can be achieved if both parameters are wild cards: _. For example: `request($var1)` $\Rightarrow$ `agent(_, _)`.

The semantics of message passing correspond to non-blocking, reliable and buffered communication. Sending a message succeeds immediately if an agent matches the definition, and the message will be stored in a buffer on the recipient. Receiving a message involves an additional unification step. The message supplied in the protocol definition is treated as a template to be matched against a message in the buffer. A unification of terms against the definition agent($\mathrm{id}_a|\ \phi$:a, $\mathrm{id}_r|\ \phi$:r) is performed. Where $\mathrm{id}_a|\ \phi$:a is matched against an agent name, or variable representing an agent name and $\mathrm{id}_r|\ \phi$:r to the agent role, or variable representing a role name. If the unification is successful, variables are bound based on the content of the message: $\phi^{(k)}$ and stored locally to the agent, for further use in the protocol. Sending will fail if no agent matches the supplied terms, and receiving will fail if no message matches the template defined in the protocol. Send and receive actions complete immediately (i.e. non blocking) and do not delay the agent.

A final sending option is provided through the multicast action. This allows an agent to broadcast the same message to all agents who have subscribed to a particular role, defined either by a role id: id_r, or term representing a role: ϕ:r

5.2.3.3 User Send and User Receive

Agents may interact directly with users by sending and receiving messages through the user action. Any data: $\phi^{(k)}$ contained in the message of performative: ρ is sent and received to and from a user. Specific information about users (such as physical network location, preferences etc.) is defined using a list configuration pairs: $config^{(k)}$.

These additional two actions allow direct interaction with a user scientist, this functionality is useful in order to keep the user in the workflow execution cycle. For example: A protocol has several execution paths but an agent cannot make a decision autonomously about which path to choose. The agent forwards these choices to the user (through user send) for an expert opinion, this series of options appears on the user's workstation. The user decides how to proceed based on the current state of the workflow and sends back the preferred execution path to the agent (via user receive).

5.2.3.4 Port Read and Port Write

As briefly mentioned in Section 5.2.2, a port is implemented as a First In First Out (FIFO) queue. Any agents within a scene can consume data from a port using the portread action, which removes the first objects from the front of the queue. The portread action is invoked with a inport name: id_{pin} or term representing an inport: ϕ:pin. Agents will read from the port k times, binding the output to local variables $\phi^{(k)}$ for use in the remainder of the protocol execution. For example: `$var1 = portread(in1)` would read from the inport: in1 to the variable: $var1. This operation is blocking, so if the port is empty then the agent will continue to wait until data becomes available.

The portwrite operation writes the terms $\phi^{(k)}$ to the outport name id_{pout} or term representing a portname ϕ:pout. As an example: `portwrite(out1, $var1)` would write the variable: $var1 to the outport: out1. The portwrite operation is non-blocking, the action completes immediately. For either action to be successful the types must be compatible.

5.2.3.5 Agent Invocation

Agents assume an initial role within a scene. However, through agent invocation, an agent can change role, introduce a new instance of a role, or make a recursive call. Agent invocation is performed by using the agent action, supplying three parameters, the first is an agent identifier: id_a or term representing an agent identifier: ϕ:a. The second is a role identifier: id_a or a term representing a role identifier: ϕ:r. The final parameter is an optional list of arguments: $\phi^{(k)}$. The agent invocation action can be used in a number of ways, by varying the parameters used:

- **Changing role:** Agents can change role during the execution of a protocol by invoking the agent action, using as parameters the same agent id: id_a and a different role id: (id_r) to their current definition. This feature avoids having to implement the same protocol code inside multiple role definitions, an agent can simply make a role switch using an agent invocation.

- **New agent:** A new instance of an agent can be instantiated by invoking the agent action with a different agent id: id_a and role id: id_r to the current definition. This feature is particularly useful if the agent has been given a task which would computationally take too long for a single agent to complete, for example extracting information from a large set of databases. In order to split up the task, new agents can be created dynamically with the agent action, using a subset of the databases as initial parameters. The number of agents generated could be decided at runtime and would be dependant on the size of the task in question.

- **Recursion:** In order to make a recursive invocation, the agent action must be called with the same agent id: id_a and role id: id_r as the current definition. Recursive calls are useful if the agent needs to repeatedly perform the same task defined by the role.

5.2.4 Operation Set

Control-flow in the protocol is enforced through the operation set, which contains a reference to the action set: α, a sequence operator: then, a choice operator: or, a parallel operator: par, an iteration operator: waitfor and a recursive operator: invoke. The operation set is illustrated by Figure 5.5 and formally represented by Figure 5.2.3.

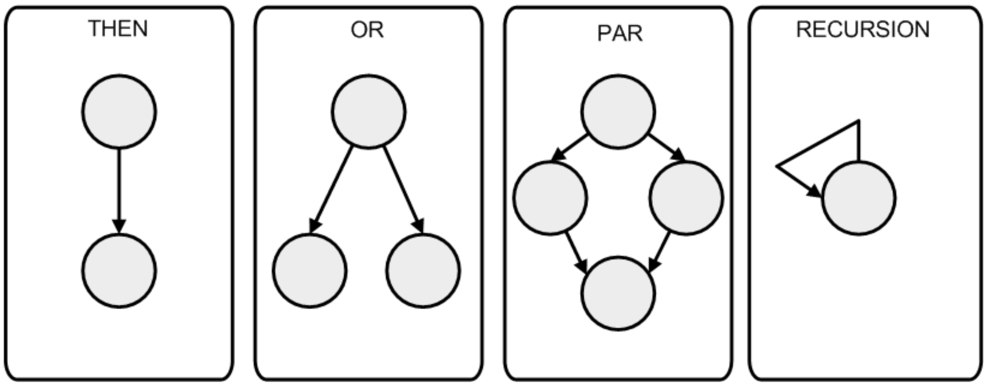


Figure 5.5: Overview of operation set

The sequence operator op_1 then op_2, evaluates op_2 only if op_1 did not contain an action that failed, otherwise it is ignored. The choice operator op_1 or op_2, handles failure in the protocol and evaluates op_2 only if op_1 contained an action that failed. The parallel operator op_1 par op_2, executes op_1 and op_2 in parallel. A waitfor loop allows repetition of sections of the protocol, nesting of the loops is possible. The body of the waitfor loop will be repeatedly executed upon failure, the loop will terminate when the loop body succeeds. If the loops times out (timeout is set with an integer value) then the actions contained within the timeout body will be executed. Timeouts allow compensation actions to be defined as they are only executed if any action inside the loop fails.

Methods can be invoked (including recursive invocations) within the protocol code, using the invoke operator. The execution engine pauses execution of the currently running method and invokes the method specified in the method identifier: id_m, or variable representing a method identifier: ϕ:m, using the parameters: $\phi^{(k)}$ as input. Once execution of the method has finished, control returns to the original method.

5.2.5 Protocol Execution

The MASC language is a specification designed to be directly executed by a group of agents. The protocol execution process is is illustrated by Figure 5.6. Once an engineer has designed a protocol describing the interaction, each agent taking part in the coordination must obtain a copy, shown as a rectangle with P inside on Figure 5.6, this copy is stored locally to each agent. Agents must then adopt a role from the role set. By adopting a role the agent must reference a reasoning web service, which implements all the decision procedures required for that role type (step 2 of Figure 5.6). This reasoning web service (marked as a rectangle with R inside) can be different for each agent and describes the agent's internal logic.

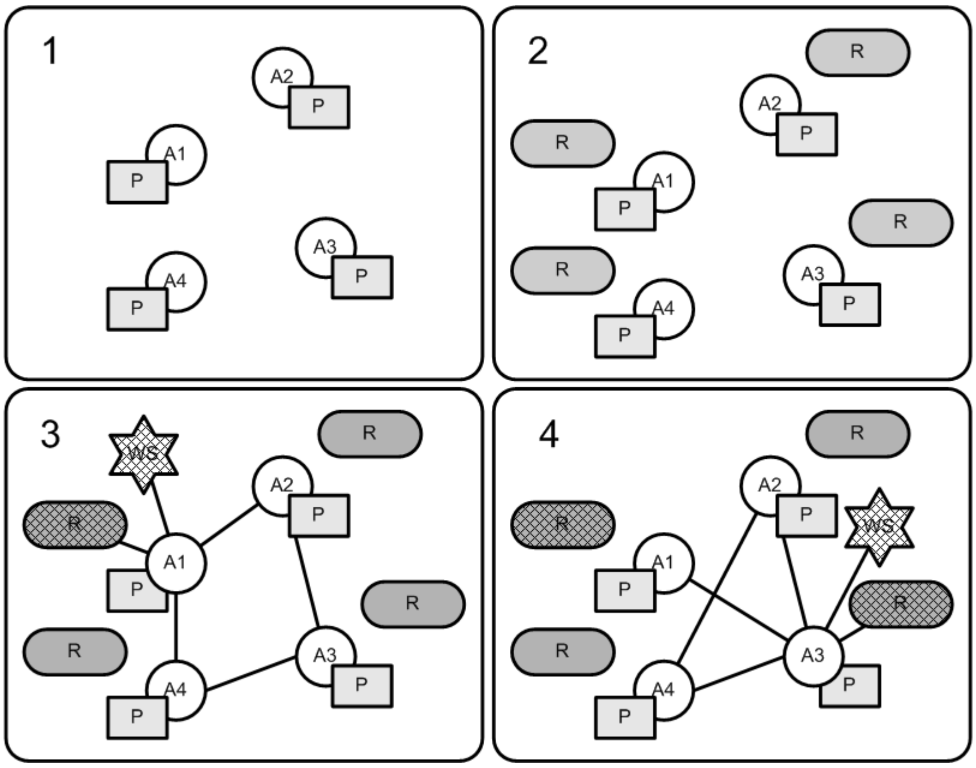


Figure 5.6: MASC protocol execution

The only requirement on an engineer designing an agent is a layer of software which can translate and execute the steps in the protocol, and a reasoning web service which implements the decision procedures of a particular role type. Each agent maintains its own internal state. This internal state records which steps of the protocol it is currently executing and any variables which may be needed for sending/receiving messages and decision procedures.

Once agents have obtained a copy of the protocol and have reference to a reasoning web service, enactment of the interaction protocol can begin. Agents follow the protocol as a script, invoking actions (from the action set) and web services if and when required. Step 3 of figure 5.6 shows a pattern of interaction taking place, with the agent in the top left invoking its reasoning web service and an external web service (illustrated by a hashed out star in Figure 5.6). A exchange of messages takes place, resulting in the agent on the bottom right invoking a method on its reasoning web service, illustrated by step 4 of figure 5.6. Execution terminates when all the protocol steps have been enacted, or the the protocol fails. Failures can be classified as *external failures*, due to faulty web services invocations; or *internal failures*, due to a badly written protocol.

5.2.6 Dataflow

The root element of the language is a protocol. With reference to Figure 5.7 a protocol is uniquely named: id_p and contains one or more scene definitions. Associated with each scene is an optional set of agents: $\{A\}$, this set can, if required be used to override the default agent configuration which a scene defines, as discussed in Section 5.2.2. For example, users may want to explicitly name agents and provide alternative implementations for the agent's decision procedures. If this set is empty the scene will be executed using the default configuration. The final association with a scene is a list of configuration pairs: $config^{(k)}$ which define any necessary configuration and startup information.

P	::=	$\texttt{protocol}(\mathrm{id}_p, \langle \mathrm{S}, \{\mathrm{A}\}, \mathrm{config}^{(k)} \rangle^{+}, \texttt{link}(\mathrm{L})^{*})$	(Protocol)
L	::=	$\mathrm{source} \rightarrow \mathrm{sink}^{+}$	(Dataflow Mapping)
source	::=	$\mathrm{linktype} \mid \texttt{outport}(\mathrm{id}_s\text{-}\mathrm{id}_{pout})$	(Dataflow Source)
sink	::=	$\mathrm{linktype} \mid \texttt{inport}(\mathrm{id}_s\text{-}\mathrm{id}_{pin})$	(Dataflow Sink)
linktype	::=	$\texttt{userinput}(\mathrm{config}^{(k)})$	(User Input)
	$\mid$	$\texttt{useroutput}(\mathrm{config}^{(k)}, \texttt{inport}(\mathrm{id}_s\text{-}\mathrm{id}_{pin})^{*})$	(User Output)
	$\mid$	$\texttt{file}(\mathrm{config}^{(k)})$	(File)
	$\mid$	ws	(Web Service)
ws	::=	$\texttt{def}(\mathrm{config}\ ^{(k)})$	(Web Service Def)

Figure 5.7: MASC formal dataflow definitions

As discussed in Section 5.2.2, a scene has a set of typed inports and outports. These port definitions allow a user to treat a scene as a composable object, through the final parameter of a protocol definition, a list of zero or more link definitions. Link definitions allow a user to compose a computational experiment by mapping a source to one or more sinks. A source can either be: an outport, user input, file input, or a web service invocation. A sink on the other hand can be one of the following: an inport, user output, file output or a web service invocation. Each of the source and sink mappings is described in more detail below:

- **Outport to Inport(s):** The most obvious mapping is from a scene's outport to one or more scene's inports. For example: `link(outport(Scene1-out1)` $\rightarrow$ `inport(Scene2:in1))` maps Scene1's outport (out1) to Scene2's inport (in1). In order for the mapping to be valid the types: τ of the outport must match the accepted types of the inport. When an agent writes to a port using the portwrite operation, the data is forwarded to source(s) which the mapping refers. When data becomes available at a scene's inport, agents can consume this data using the portread operation, discussed in Section 5.2.3.4.

- **User Interaction:** User interaction in relation to agents was discussed in Section 5.2.3.3 through the sending and receiving of messages to users. Within the dataflow layer user interaction is handled in two ways, by mapping a user to a scene's inport(s), or mapping a scene's outport(s) to a user. Mapping user interaction to a scene's inport is achieved through a link definition where a userinput is the source and an inport is the sink. In this instance a user must supply typed data which matches the inport definition: inport(id_s:id_{pin}, τ, boolean), illustrated by Section 1 of Figure 5.8. Output from a scene can also be mapped to a user, by supplying a link where the source is an outport and the sink is a useroutput definition. Here the output can be mapped directly to the user (if the final parameter is empty), or to a selection of inports which the user has control over. Mappings of the latter kind give the user direct control over the dataflow, allowing them to select which port(s) to write the scene's output data to. Section 2 of Figure 5.8 illustrates a scene which is attempting to write data to one of two ports, the data however is forwarded to the user, who decides it should be written to the scene on the far right.

- **Web Services and Files:** Users can supply a mapping where a source is a web service or file and the sink is a scene's inport. In this instance the output from the web service invocation or data read in from a file is written to the scene's inport.

This process can also work the other way round, a user can supply a mapping where a source is an outport and the sink is a file or web service, allowing agents to output data to external sources.

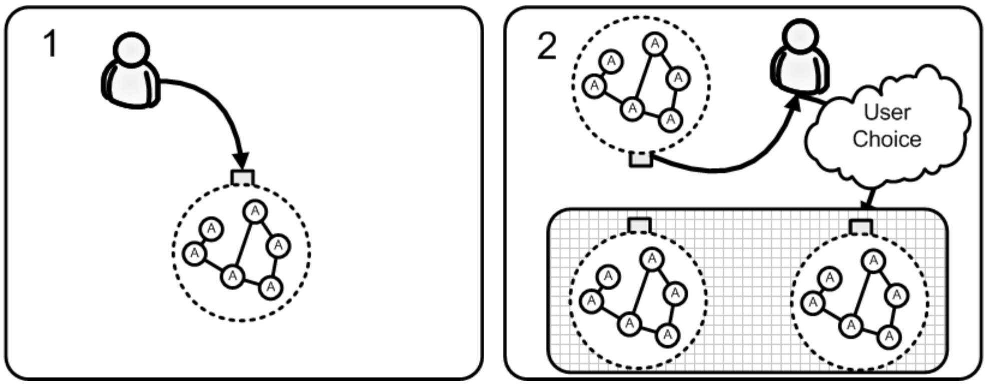


Figure 5.8: MASC user interaction

There are several restrictions placed on the mappings a user can make through the MASC language, this is summarised by Table **??**.

Valid Source to Sink Mappings		
Source		**Sink**
outport(id$_s$-id$_{pout}$)	$\rightarrow$	inport(id$_s$-id$_{pin}$)
userinput(config$^{(k)}$)	$\rightarrow$	inport(id$_s$-id$_{pin}$)
file(config$^{(k)}$)	$\rightarrow$	inport(id$_s$-id$_{pin}$)
def(config$^{(k)}$)	$\rightarrow$	inport(id$_s$-id$_{pin}$)
outport(id$_s$-id$_{pout}$)	$\rightarrow$	useroutput(config$^{(k)}$,inport(id$_s$-id$_{pin}$)*)
outport(id$_s$-id$_{pout}$)	$\rightarrow$	file(config$^{(k)}$)
outport(id$_s$-id$_{pout}$)	$\rightarrow$	def(config$^{(k)}$)

Table 5.2: MASC valid dataflow mappings

Figure 5.9 illustrates graphically the concept of treating scenes as composable objects to form higher level computational experiments, in this example five scenes are wired together, taking input from a user and producing output to files. Scenes are effectively treated as parameterisable patterns of interaction, it is then up to the user to wire together these black boxes by supplying the dataflow mapping, through a set of links.

Scenes begin the process of execution described in Section 5.2.5 when all core inports have been written. Core inports are identified by setting the final boolean parameter to true. As discussed in Section 3.4 scientific workflows tend to have an execution model which emphasises dataflow. Port definitions allow a scene to be treated as a composable object, allowing our agent-based interaction model to fit in with the dataflow paradigm used by most scientific workflow modelling tools. For example a pattern of interaction expressed as a scene could be treated as a node in a scientific workflow graphical composition tool, such as Taverna. This allows techniques from the multiagent systems community to be seamlessly integrated into the existing architecture.

Through the addition of a dataflow layer, scientists can treat scenes simply as parameterisable black boxes of computation, without getting involved with the messy details of concurrent protocol design. This is a useful abstraction mechanism and allows an experiment to be constructed at a higher level by specifying a set of links which wire the experiment execution together.

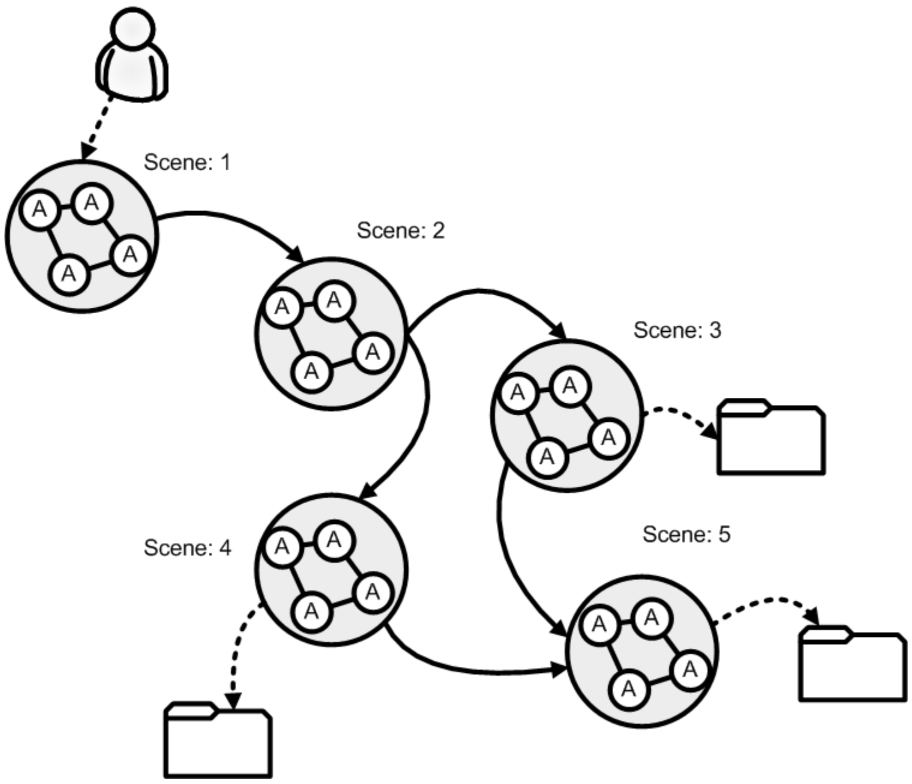


Figure 5.9: Example dataflow mapping

5.3 Chapter Conclusions

Our requirements analysis is based on the review of existing scientific workflow systems, AstroGrid workflow scenarios and a counterexample scenario taken from the LSST project. Through this detailed requirements analysis this thesis has identified the need for flexible, ad-hoc service composition. In order to meet these requirements, this Chapter has presented an agent-based solution to the service composition problem, providing flexible, runtime coordination of services. Our approach is founded on the concept of interaction protocols and facilitated through the MultiAgent Service Composition (MASC) language. This Chapter has presented in detail the MASC language syntax and explained the reasons for the choices made, providing where necessary simple examples. The following Chapter presents a full implementation of the MASC language, through an open-source Java-based web service composition tool: `Zorro`.

5.4 Complete MASC Language Syntax

P	::=	$\texttt{protocol}(\mathrm{id}_p, \langle S, \{A\}, \mathrm{config}^{(k)}\rangle^+, \texttt{link}(L)^*)$	(Protocol)
S	::=	$\texttt{scene}(\mathrm{id}_s, \{R\}, \{A\}, \{\mathrm{inport}\}, \{\mathrm{outport}\})$	(Scene)
A	::=	$\texttt{agent}(\mathrm{id}_a, \mathrm{id}_r, \phi^{(k)})$	(Agent)
R	::=	$\langle \mathrm{id}_r, \mathrm{config}^{(k)}, \{M\}\rangle$	(Role)
M	::=	$\texttt{method}\ \mathrm{id}_m \mid \phi{:}\mathrm{m}(\phi^{(k)}) = \mathrm{op}$	(Method)
op	::=	α	(Action)
	$\mid$	$\mathrm{op}_1\ \texttt{then}\ \mathrm{op}_2$	(Sequence)
	$\mid$	$\mathrm{op}_1\ \texttt{or}\ \mathrm{op}_2$	(Choice)
	$\mid$	$\mathrm{op}_1\ \texttt{par}\ \mathrm{op}_2$	(Parallel Composition)
	$\mid$	$\texttt{waitfor}\ \mathrm{op}_1\ \texttt{timeout}\ \mathrm{op}_2$	(Iteration)
	$\mid$	$\texttt{invoke}\ \mathrm{id}_m \mid \phi{:}\mathrm{m}(\phi^{(k)})$	(Recursion)
α	::=	ε	(No Action)
	$\mid$	proc	(Procedure)
	$\mid$	$\texttt{agent}(\mathrm{id}_a \mid \phi{:}\mathrm{a}, \mathrm{id}_r \mid \phi{:}\mathrm{r}, \phi^{(k)})$	(Agent Invocation)
	$\mid$	$\rho(\phi^{(k)}) \Rightarrow \texttt{agent}(\mathrm{id}_a \mid \phi{:}\mathrm{a}, \mathrm{id}_r \mid \phi{:}\mathrm{r})$	(Send)
	$\mid$	$\rho(\phi^{(k)}) \Rightarrow \texttt{multicast}(\mathrm{id}_r \mid \phi{:}\mathrm{r})$	(MultiCast)
	$\mid$	$\rho(\phi^{(k)}) \Rightarrow \texttt{user}(\mathrm{config}^{(k)})$	(User Send)
	$\mid$	$\rho(\phi^{(k)}) \Leftarrow \texttt{agent}(\mathrm{id}_a \mid \phi{:}\mathrm{a}, \mathrm{id}_r \mid \phi{:}\mathrm{r})$	(Receive)
	$\mid$	$\rho(\phi^{(k)}) \Leftarrow \texttt{user}(\mathrm{config}^{(k)})$	(User Receive)
	$\mid$	$\phi^{(k)} = \texttt{portread}(\mathrm{id}_{pin} \mid \phi{:}\mathrm{pin})$	(Port Read)
	$\mid$	$\texttt{portwrite}(\mathrm{id}_{pout} \mid \phi{:}\mathrm{pout}, \phi^{(k)})$	(Port Write)
proc	::=	$\neg\ \mathrm{proc} \mid \mathrm{proc} \wedge \mathrm{proc} \mid \mathrm{proc} \vee \mathrm{proc}$	
	$\mid$	$\phi^{(k)} = \rho(\phi^{(l)})\ \texttt{fault}\ \phi^{(m)}$	(Decision Procedure)
	$\mid$	$\phi^{(k)} = \texttt{service}(\mathrm{ws}^+, \phi^{(l)})\ \texttt{fault}\ \phi^{(m)}$	(Web Service Invocation)
ϕ	::=	$\mathrm{v}{:}\tau \mid \_ \mid \mathrm{c}{:}\tau$	(Terms)
L	::=	$\mathrm{source} \longrightarrow \mathrm{sink}^+$	(Dataflow Mapping)
source	::=	$\mathrm{linktype} \mid \texttt{outport}(\mathrm{id}_s\text{-}\mathrm{id}_{pout})$	(Dataflow Source)
sink	::=	$\mathrm{linktype} \mid \texttt{inport}(\mathrm{id}_s\text{-}\mathrm{id}_{pin})$	(Dataflow Sink)
inport	::=	$\texttt{inport}(\mathrm{id}_s\text{-}\mathrm{id}_{pin}, \tau, \mathrm{boolean})$	(Inport Definition)
outport	::=	$\texttt{outport}(\mathrm{id}_s\text{-}\mathrm{id}_{pout}, \tau)$	(Outport Definition)
linktype	::=	$\texttt{userinput}(\mathrm{config}^{(k)})$	(User Input)
	$\mid$	$\texttt{useroutput}(\mathrm{config}^{(k)}, \texttt{inport}(\mathrm{id}_s\text{-}\mathrm{id}_{pin})^*)$	(User Output)
	$\mid$	$\texttt{file}(\mathrm{config}^{(k)})$	(File)
	$\mid$	ws	(Web Service)
ws	::=	$\texttt{def}(\mathrm{config}\ ^{(k)})$	(Web Service Def)
config	::=	$\langle \mathrm{name}, \mathrm{value}\rangle$	(Configuration Pair)

Chapter 6

An Agent-Based Web Services Composition Framework

The `Zorro` framework is an agent-based web services composition tool founded on the Multi Agent Service Composition (MASC) language. This framework is an open-source Java implementation and has served as a test bed for the ideas addressed by this thesis, allowing real protocols to be executed with real services on real data.

This Chapter discusses the implementation and algorithms of the Zorro framework in detail. Section 6.1 presents an overview of the technologies used as part of the implementation, in particular how the formal MASC syntax is represented and manipulated. Section 6.2.1 describes the architecture of a generic coordination service which is capable of dynamically unmarshalling a scene definition and building an internal representation for execution. In Section 6.2.2 the process of creating and initialising agents to enact a workflow is described, with a simple XML example. Section 6.2.3 discusses the architecture of individual agents and how a workflow is enacted by a group of distributed agents. Section 6.3 describes how scenes are composed into more complex workflows through the dataflow layer. Finally, conclusions are presented in Section 6.4.

6.1 MASC Language Representation

A combination of technologies have been used to represent, parse and execute the MASC language, discussed in Chapter 5. The first of these technologies is the *Java Web Services Development Pack (JWSDP)* [49], which is an integrated toolkit, allowing developers to build and test XML applications, web services, and web applications. This framework is made up of many interconnected components, however this Chapter will focus on the two which have been utilised by the implementation framework: The *Java Architecture for XML Binding (JAXB)* and the *Java API for XML Based Remote Procedure Call (JAX-RPC)*, which will be addressed by Section 6.2.3.

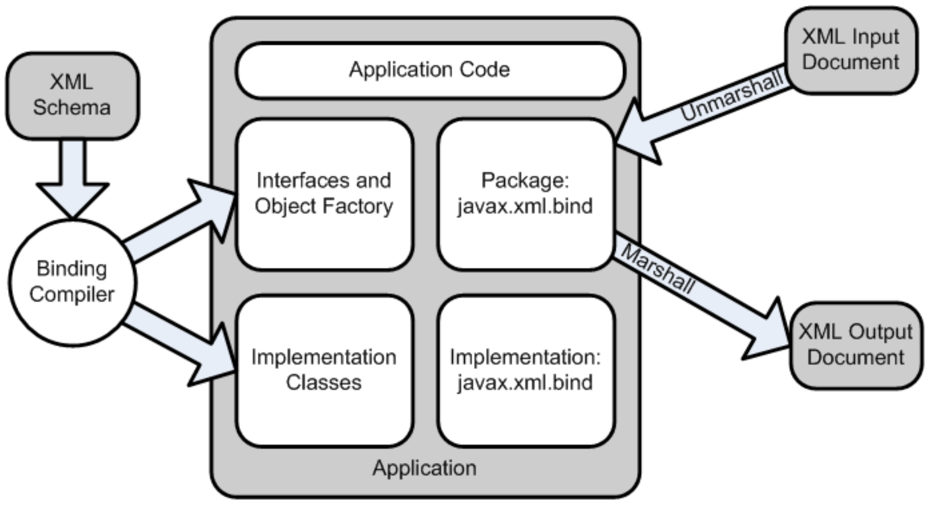


Figure 6.1: JAXB architecture overview

The JAXB architecture allows an XML Schema definition to be bound to concrete Java classes, allowing developers to incorporate XML data and processing functions into their applications. As part of this process, JAXB provides methods for unmarshalling XML instance documents to a Java Content Tree, and marshalling a Java Content Tree back into XML instance documents. The JAXB architecture is aimed as a replacement for older XML processing technologies such as SAX and DOM [47]. Figure 6.1 illustrates the core components in the JAXB model, which will be explained in more detail below:

- **XML Schema:** The XML Schema Definition Language (XSD) [27] is a W3C recommendation and one of many XML schema languages. XSD is used to express a set of rules which define the legal building blocks of an XML document, typically expressed in terms of constraints on the structure and content of documents: elements, attributes, data types etc.

- **Binding compiler:** The JAXB binding compiler is the core of the JAXB processing model. Its function is to transform, or bind, a source XML schema to a set of JAXB content classes in the Java programming language. When executed using an XML schema as input (optionally with custom binding declarations) the binding compiler generates Java classes that map to constraints in the source XML schema.

- **Java application:** In the context of JAXB, a Java application is a client that uses the JAXB binding framework to unmarshal XML data, validate and modify it, marshalling the Java content back to XML data.

- **XML input documents:** XML content can be unmarshalled by converting an XML instance document to an internal representation using a Java Content Tree. Once an XML instance is unmarshalled it can then be manipulated, marshalling an updated version if necessary. Validation of an XML instance document against the source schema is supported, forcing strict adherence to an XML schema.

- **XML output documents:** The process described above can also work in the opposite direction, XML (internally represented as a Java Content Tree) can be marshalled to an XML document. Marshalling involves parsing the internal representation and writing an XML document that is an accurate representation, and valid with respect to the source XML schema.

The MASC language has been represented using an XML Schema, providing a straight forward conversion from the formal syntax to a computer interpretable form. The full XML Schema definition can be found in Appendix A. Skeleton parser code has been generated by providing the JAXB compiler with the MASC XML Schema as input, following the process described above. There are a number of components in the framework which utilise this parsing component, these will be discussed in more detail by the following Sections. The type system in the Zorro framework is mapped to the standard set of JAX-RPC supported types: Boolean, Byte, Double, Float, Integer, Long, Short, String, (Arrays and multidimensional arrays are also supported).

6.2 Scene Implementation

A scene acts as a closely coupled system and is responsible for initialising and controlling the execution of a group of agent roles which execute a scene specification. Communication between scenes takes place through a WSDL interface, allowing it to be deployed anywhere on the network and be treated as a composeable object to form more complex workflows, this will be discussed in more detail in Section 6.3. Figure 6.2 illustrates an overview of a scene's component model. A scene contains a number of interacting components which will each be discussed in turn by the following sub sections.

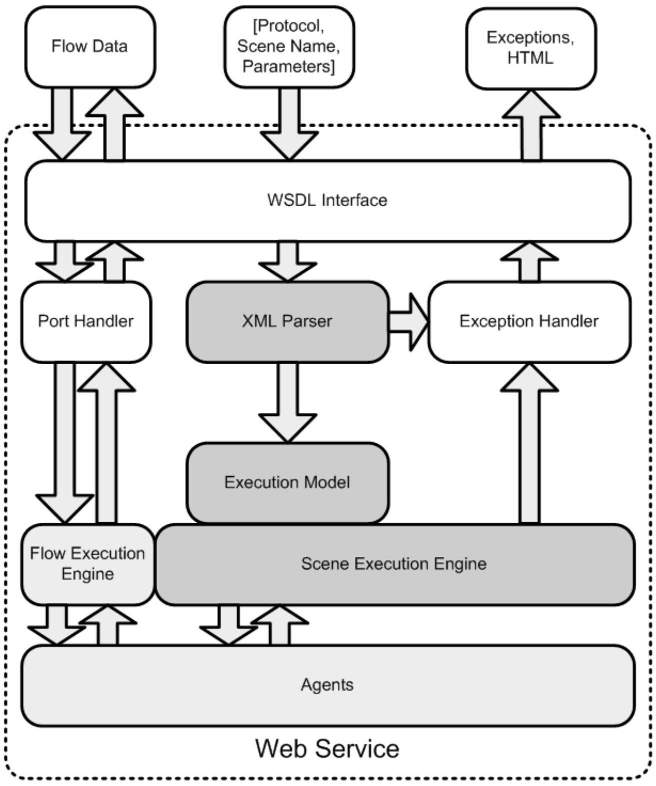


Figure 6.2: Scene architecture overview

6.2.1 Building the Execution Model and Resolving Dependencies

A scene of computation is executed by a *coordination service*. A coordination service is a simple, lightweight layer of software which translates and executes a scene definition. It is a generic service and can execute any valid definition. Figure 6.3 illustrates a scene's initialisation algorithm, with reference to this Figure and Figure 6.2 the following process takes place:

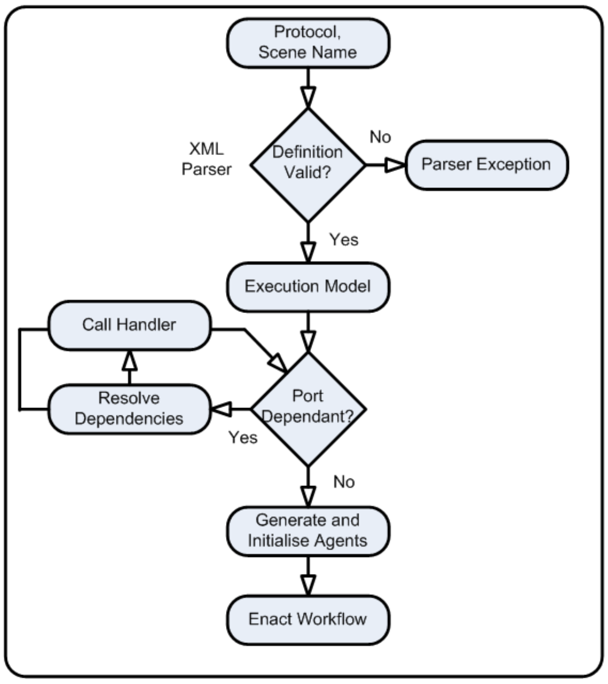


Figure 6.3: Scene Initialisation algorithm

- **Receive request:** A coordination service is initialised when it receives a Remote Procedure Call (RPC) invocation containing a protocol (XML instance), and the name of a scene within that protocol it is required to adopt.
- **JAXB XML parser:** Once a protocol is received, the parsing component will unmarhsall and validate it against the MASC XML Schema. Any exceptions through a malformed protocol are thrown to the exception handler, initialisation is terminated and exceptions are reported to the user.
- **Execution model:** If the validation is successful, the XML parser (implemented through JAXB) converts the scene definition (represented as XML) to an internal execution model. This internal execution model is represented as a Java Content Tree and allows manipulation of the scene definition.
- **Build ports:** Before initialisation and execution of the agents can begin, the flow execution engine must check if the scene is part of a more complex workflow, defined through a dataflow mapping. If this is the case then the flow execution engine must dynamically build any ports which are part of the scene definition and resolve any dependencies these ports may have.

- **Resolve dependencies:** Dependencies are resolved by parsing the set of links (defined in the protocol mapping) and retrieving those where the sink is an inport definition belonging to the current scene and the core value is set to true (must be written before scene execution can begin). If dependencies exist on any of the scene's inports the flow engine must call the appropriate handler. Handlers have been implemented which allow web service invocation, file manipulation and user interaction. The handler will retrieve the necessary data (by reading from a file, interacting with a user etc.) forwarding it the required inport. Dataflow information (shown as flow data in Figure 6.2) is sent and received through the coordination service's WSDL interface. However, if the scene has no dependencies initialisation of the agents can take place.

6.2.2 Initialisation of Agents

Once any port dependencies have been resolved, dynamic initialisation of the agents can begin, this is handled by the *Scene Execution Engine* component shown on Figure 6.2. Agents can be initialised through a default setting or a customised setting. To illustrate this point, Figure 6.4 represents an example of use, for simplicity only the necessary protocol features are included.

The syntax contains a scene, `scene1` and within that scene there are two role definitions, a `coordinator` and a `participant`. As discussed by the previous Chapter, role definitions provide a generic pattern of interaction. Agents adopt roles from a scene and decision procedures provide the hook from a role definition to a particular, grounded model of agency. With reference to the example, by default a `coordinator` role's reasoning service is located at `http://location1?WSDL` (line 1 of Figure 6.4) and a `participant` role's reasoning service is located at `http://location2?WSDL` (line 2 of Figure 6.4).

Agents must then adopt roles from a scene definition. In our example a user has explicitly created an agent which will adopt the `coordinator` role (lines 3-6 of Figure 6.4). Here the default settings of the role have been overwritten, the location of the reasoning service has been changed to `http://location3?WSDL`, the name of the agent is set to `myCoordinator` and details of how long the agent will wait to receive a port message (`PortWait`) and from another agent (`RecvWait`) have been set. In order to interact with a `coordinator`, two agents adopting the `participant` role (lines 7-8 of Figure

```
<scene name="scene1">
    <!-- Coordinator Role Definition -->
1   <agent implementation="http://location1?WSDL"
     max="1" min="1" role="coordinator">
        <method> ... </method>
    </agent>

    <!-- Participant Role Definition -->
2   <agent implementation="http://location2?WSDL"
     max="undefined" min="1" role="participant">
        <method> ... </method>
    </agent>
    ...
</scene>

<mapping name="demomapping">
    <node location="" name="scene1">
        <!-- Customised Coordinator -->
        <role name="coordinator">
3          <agent implementation="http://location3?WSDL"
4           name="myCoordinator" num="1"
5           portwait="10"
6           recvwait="10"/>
        </role>

        <!-- Customised Participants -->
        <role name="participant">
7          <agent implementation="http://location4?WSDL"
            name="myParticipant1" num="1"
            portwait="10"
            recvwait="10"/>
        </role>

        <role name="participant">
8           <agent implementation="http://location5?WSDL"
            name="myParticipant2" num="1"
            portwait="10"
            recvwait="10"/>
        </role>
    </node>
    <link>...</link>
</mapping>
```

Figure 6.4: Sample XML protocol - initialising agents

6.4) have been created: `MyParticipant1` and `Myparticipant2`, each referencing a different reasoning service.

After parsing the scene definition, creating the execution model and resolving any port dependencies the scene execution engine creates a separate thread for each agent, our example in Figure 6.4 would generate one `coordinator` agent and two `participants`. Each agent has a local copy of the protocol and is independently capable of parsing and executing the protocol. Agents maintain internal state, this internal state records which steps of the protocol it is currently executing and any variables which may be needed for sending/receiving messages and decision procedures.

6.2.3 Enacting the Workflow

Agents act as peers, forming a *peer-to-peer system*. As each agent has a local copy of the protocol, no centralised control is required. Once all agents have been created and initialised, enactment of the workflow can begin. Each agent follows the role definition like a script, calling the necessary actions when specified by the protocol. An overview of the components making up the agent architecture is illustrated by Figure 6.5.

External and reasoning service invocations are handled by the JAX-RPC interface. JAX-RPC is a technology for building web services and clients that use remote procedure calls (RPC) and XML. Often used in a distributed client-server model, an RPC mechanism enables clients to execute procedures on other systems. In JAX-RPC, a remote procedure call is represented by an XML-based protocol such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing remote procedure calls and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

When an agent is required to execute an action, the appropriate handler is invoked, the process for each handler is described below:

- **Decision procedure invocation:** When an agent is required to execute a service (either a decision procedure or an external service), the following process takes place, this is illustrated by the algorithm displayed in Figure 6.6. Firstly an agent parses the decision procedure definition in the protocol, consisting of the: procedure name, input parameters and output parameters. This protocol definition is then compared to WSDL definition located in the agent's reasoning web service.

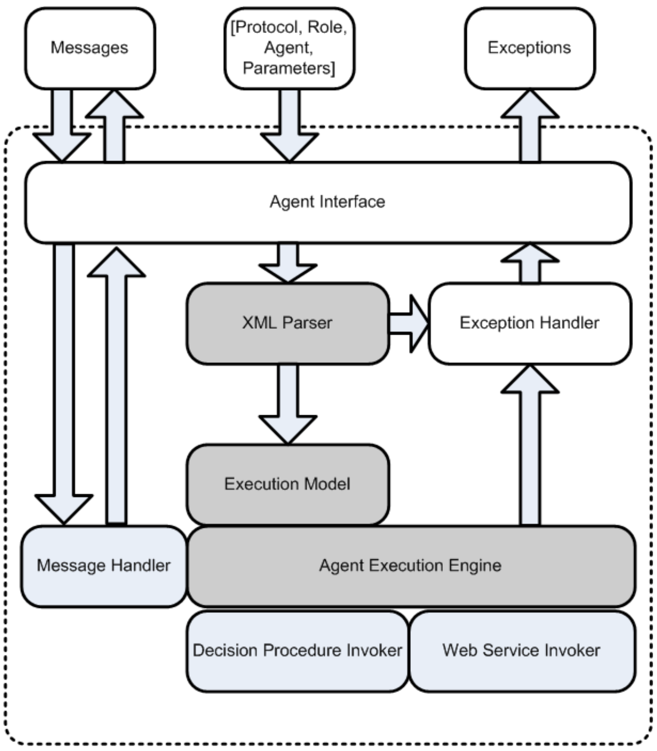


Figure 6.5: Agent architecture

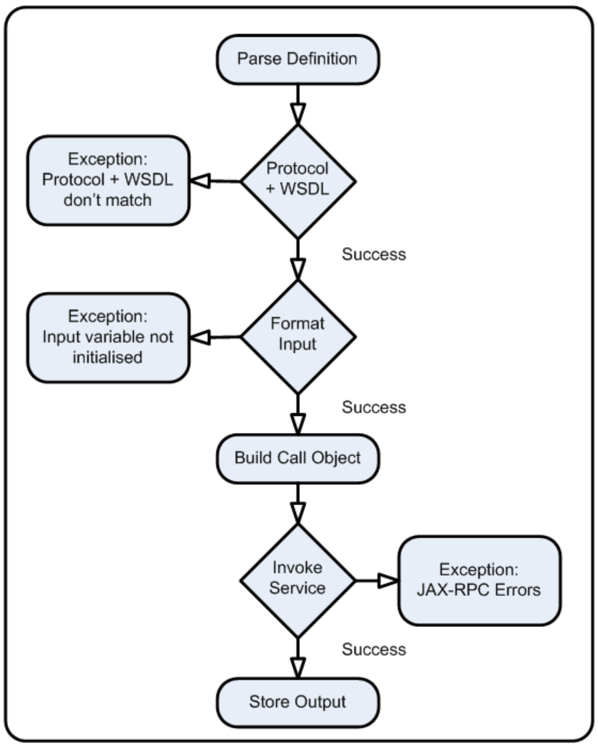


Figure 6.6: Service invocation algorithm

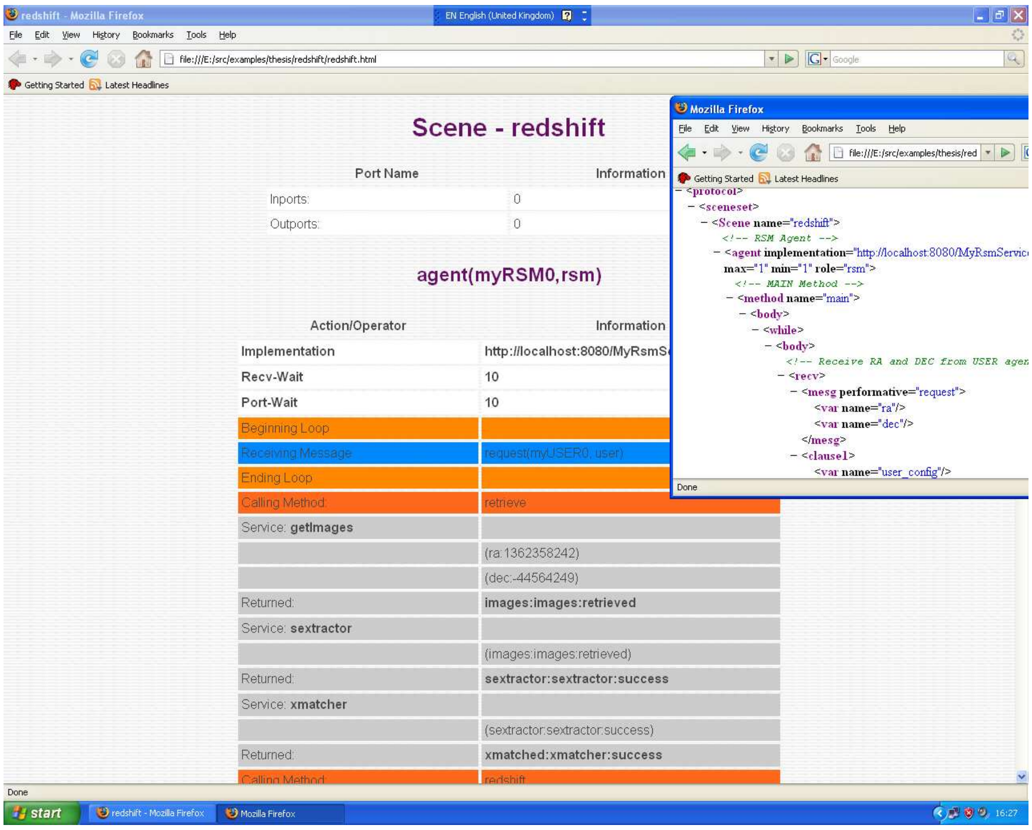
This comparison utilises the WSDL4J interface [70]. If there are any inconsistencies, such as the: wrong number of input/output parameters, wrong type of input/output parameters, wrong method name etc. then an exception is thrown and backtracking of the protocol begins. However, if the protocol matches the WSDL definition then an agent can begin to format the input for the invocation. This is achieved by retrieving any variables (local or from a scene) that are required to be used as input. Execution will terminate and backtracking of the protocol will begin if variables haven't been initialised, types don't match etc. However, if successful a JAX-RPC Call object is constructed and the decision procedure is invoked on the agent's reasoning service. Exceptions caused as a result of an invocation are labelled as JAX-RPC exceptions. Any output from the invocation is stored locally to the agent, by updating existing variables or creating new ones.

- **External service invocation:** This process is similar to the decision procedure invocation. However, instead of invoking a method on the agent's reasoning web service an external service is called. This handler is generic, it can call any method once is has obtained the WSDL definition. Firstly an agent retrieves the WSDL document, as specified by the service definition (this can be hard-coded or represented as a variable at runtime). A check is then made to ensure that all the details in the protocol definition match those in the WSDL definition, comparisons are made against: the number and type of input/output parameters, namespace, operation name, port name and service name. If these details are not consistent then an exception is thrown and backtracking of the protocol begins. If this process is successful then an agent can follow the same steps as described for the decision procedure invocation: formatting the input, building a call object, invoking a service and storing output variables.

- **Message passing:** Each agent runs as a separate thread within a scene execution engine. When an agent is required to send a message to another agent, the input message queue on the recipient agent is locked and the message is passed between threads. The recipient agent can then check its input queue, utilising the message content when required.

- **Port reading/writing:** If the scene is part of a more complex workflow then agents can read and write to ports. If an agent is reading data from an input port, it is removed in a First In First Out (FIFO) fashion and is stored locally to the agent. Agents can also write data to output ports, which is forwarded to the port's sink by the appropriate handler (scene, web service, file, user interaction etc.).

Execution of a scene terminates when all the protocol steps have been enacted, or the the protocol fails. Failures can be classified as *external failures*, due to faulty web services invocations; or *internal failures*, due to a badly written protocol. Each agent operating within a scene outputs a text log file, each log file is concatenated to form a scene description which is formatted in html. This allows a user of the system to view exactly how the protocol has executed. An example of the output is illustrated by Figure 6.7.


redshift - Mozilla Firefox
EN English (United Kingdom)
File Edit View History Bookmarks Tools Help
file:///E:/src/examples/thesis/redshift/redshift.html
Getting Started Latest Headlines
Done
start
redshift - Mozilla Firefox
Mozilla Firefox
16:27

Scene - redshift

Port Name	Information
Inports:	0
Outports:	0

agent(myRSM0,rsm)

Action/Operator	Information
Implementation	http://localhost:8080/MyRsmS
Recv-Wait	10
Port-Wait	10
Beginning Loop	
Receiving Message	request(myUSER0, user)
Ending Loop	
Calling Method:	retrieve
Service: **getImages**	
	(ra:1362358242)
	(dec:-44564249)
Returned:	**images:images:retrieved**
Service: **sextractor**	
	(images:images:retrieved)
Returned:	**sextractor:sextractor:success**
Service: **xmatcher**	
	(sextractor:sextractor:success)
Returned:	**xmatched:xmatcher:success**
Calling Method:	redshift

Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///E:/src/examples/thesis/red
Getting Started Latest Headlines
Done

```
- <protocol>
  - <sceneset>
    - <Scene name="redshift">
        <!-- RSM Agent -->
      - <agent implementation="http://localhost:8080/MyRsmServic
        max="1" min="1" role="rsm">
          <!-- MAIN Method -->
        - <method name="main">
          - <body>
            - <while>
              - <body>
                  <!-- Receive RA and DEC from USER agen
                - <recv>
                  - <mesg performative="request">
                      <var name="ra"/>
                      <var name="dec"/>
                    </mesg>
                  - <clause1>
                      <var name="user_config"/>
```

Figure 6.7: Sample execution output

6.3 Composing Scenes into More Complex Workflows

Scenes of computation can be executed independently or part of a more complex workflow. A user interested in composing and executing multiple workflow components can approach the Zorro framework from the dataflow layer. From this level of abstraction, an engineer does not need to concern themselves with the intricate details of protocol design, scenes can be simply be treated as composeable objects. Figure 6.8 illustrates the initialisation algorithm for an entire protocol, consisting of multiple workflow components.

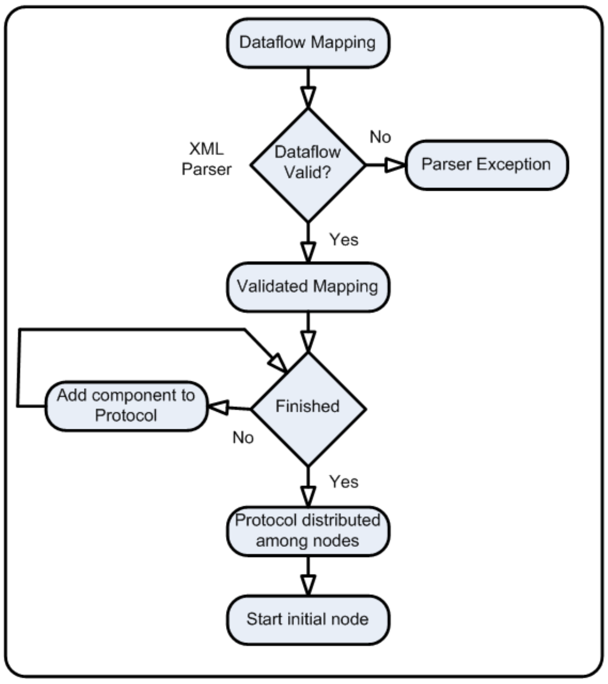


Figure 6.8: Protocol execution algorithm

This process consists of the following steps:

- **Validate dataflow mapping:** In order to build a protocol, a user supplies a dataflow mapping of workflow components. This mapping is validated against the MASC XML Schema definition, if the instance is not valid in respect to the source schema then an exception is thrown and the user must go about redesigning the mapping.

- **Dynamically build protocol:** The mapping supplied is simply a description of the components required in the workflow, specifying how these components interact with one another, through dataflow. Before execution can begin, the framework must dynamically build a complete description of the workflow (based on the dataflow description), splicing in all the necessary components to form an executable protocol. It is important to note that these components could be distributed (for example scene descriptions kept in a remote repository), therefore the framework must retrieve each component before execution of the protocol can begin.

- **Execute protocol:** Once all components have been spliced into the protocol, execution of the workflow can begin. The protocol can be executed locally or disseminated to multiple, distributed coordination services, this is the user's choice and several options exist for configuring the system. Once every node has a complete copy of the protocol, enactment begins. Scenes execute by following the same process as described by Section 6.2, beginning execution when all of the inports have been satisfied. To illustrate this point, consider a simple extension to our earlier example, the XML syntax for the syntax is displayed in Figure 6.9. A user has configured two scenes to be executed as part of a workflow: `scene1` and `scene2`. These scene definitions are remote and must be retrieved and spliced into the protocol before execution can being. The user has supplied a mapping between these scenes, mapping the outport of scene1: `scene1_out1` to the inport of scene2: `scene2_in1`. When execution begins, scene1 will start immediately as there are no dependent ports, scene2, however enters a wait state, beginning execution when data is written from scene1's outport to scene2's inport.

```
<protocol>
    <scene name="scene1">
        <!-- Output port definition -->
        <output>
            <port name="scene1_out1" type="xsd:string"/>
        </output>
        ...
    </scene>

    <scene name="scene2">
        <!-- Input port definition -->
        <input>
           <port name="scene2_in1" type="xsd:string" core="true"/>
        </input>
        ...
    </scene>

    <!-- User’s mapping -->
    <mapping name="demomapping">

        <!-- Nodes to be included -->
        <node location="http://location1" name="scene1"/>
        <node location="http://location2" name="scene2"/>

        <!-- Link definition -->
        <link>
            <source>
                <outport port="scene1_out1" scene="scene1"/>
            </source>
            <sink>
                <inport port="scene2_in1" scene="scene2"/>
            </sink>
        </link>
    </mapping>
</protocol>
```

Figure 6.9: Sample XML protocol - dataflow mapping

6.4 Chapter Conclusions

This Chapter has presented the Zorro framework, an agent-based web services composition tool to enact distributed scientific workflows. This framework has helped bring to life the ideas addressed by this thesis, allowing protocols to be executed on live services and data. The implementation is open-source and available for download from: `http://www.mas.sourceforge.net`. The following Chapter presents a methodology for building systems using our approach, a term we label *coordination-oriented programming* and demonstrates, by example how the MASC language meets the original set of requirements and solves the set of motivating workflow scenarios presented in Sections 3.3, 3.2 and 4.1.

Chapter 7

Evaluation By Use-Case

This Chapter ties together all of the separate sections of the thesis, demonstrating how our agent-based approach to service composition (MASC) can solve the original set of workflow scenarios and meet the requirements of scientific workflow, addressed by Chapters 3, 4 and 5. Section 7.1 proposes the coordination-oriented programming methodology, outlining how users can build complex distributed systems using the techniques addressed by this thesis. This methodology describes how users can approach the system from different levels of abstraction, adopting the role of either an: *interaction engineer* (Section 7.1.1), *experiment engineer* (Section 7.1.2) or an *agent engineer* (Section 7.1.3).

This methodology is then applied to each of the motivating workflow scenarios, demonstrating how the various stages of the coordination-oriented programming methodology builds up a working protocol to solve the original specification. Each Section will provide a full implementation using the abstract MASC language and discus how this implementation was realised, outlining why certain choices regarding language features were made. Alongside the abstract syntax is a concrete XML specification which has been deployed on the Zorro framework, each implementation is contained in the relevant appendix. Section 7.2.1 discusses the batch processing scenario, Section 7.2.2, the knowledge acquisition scenario and finally Section 7.2.3 addresses the runtime coordination scenario.

Section 7.3 addresses a number of important points, firstly how our hybrid multiagent system/service-oriented architecture approach to service composition fulfills the original requirements. Secondly, how our approach can solve a new class of workflow, involving flexible, runtime service composition. Finally, this Section discusses the advantages and disadvantages of our approach in relation to existing workflow composition languages and tools. Conclusions are presented in Section 7.4

7.1 Coordination-Oriented Programming Methodology

In addition to providing the MASC language and Zorro framework for scientific workflow composition, we propose a methodology outlining how users can build a workflow which solves a specification using our agent-based approach to service composition. In order to allow users with different skills and motivations to take advantage of the MASC language it can be approached from various levels of abstraction, dividing users into three distinct categories: *experiment engineers*, *interaction engineers* and *agent engineers*, this concept is illustrated by Figure 7.1. Each of these levels of abstraction will now be addressed by the following Sections.

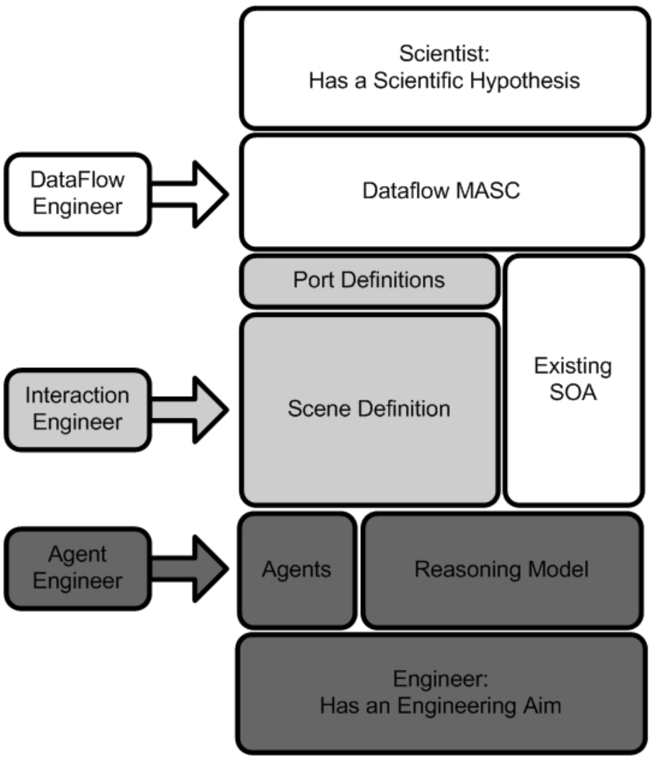


Figure 7.1: MASC layers of abstraction

7.1.1 Interaction Engineer

Interaction engineers as the name suggests, are primarily concerned with coordination. Interaction engineers take a software specification and divide it into a number of distinct agent roles, specifying the details of how these roles coordinate with one another (within a multiagent system) to achieve the overall aim of the specification. Using the features provided by the MASC language interaction engineers build Scenes, consisting of roles which are themselves constructed using the action and operation sets discussed in Sections 5.2.3 and 5.2.4.

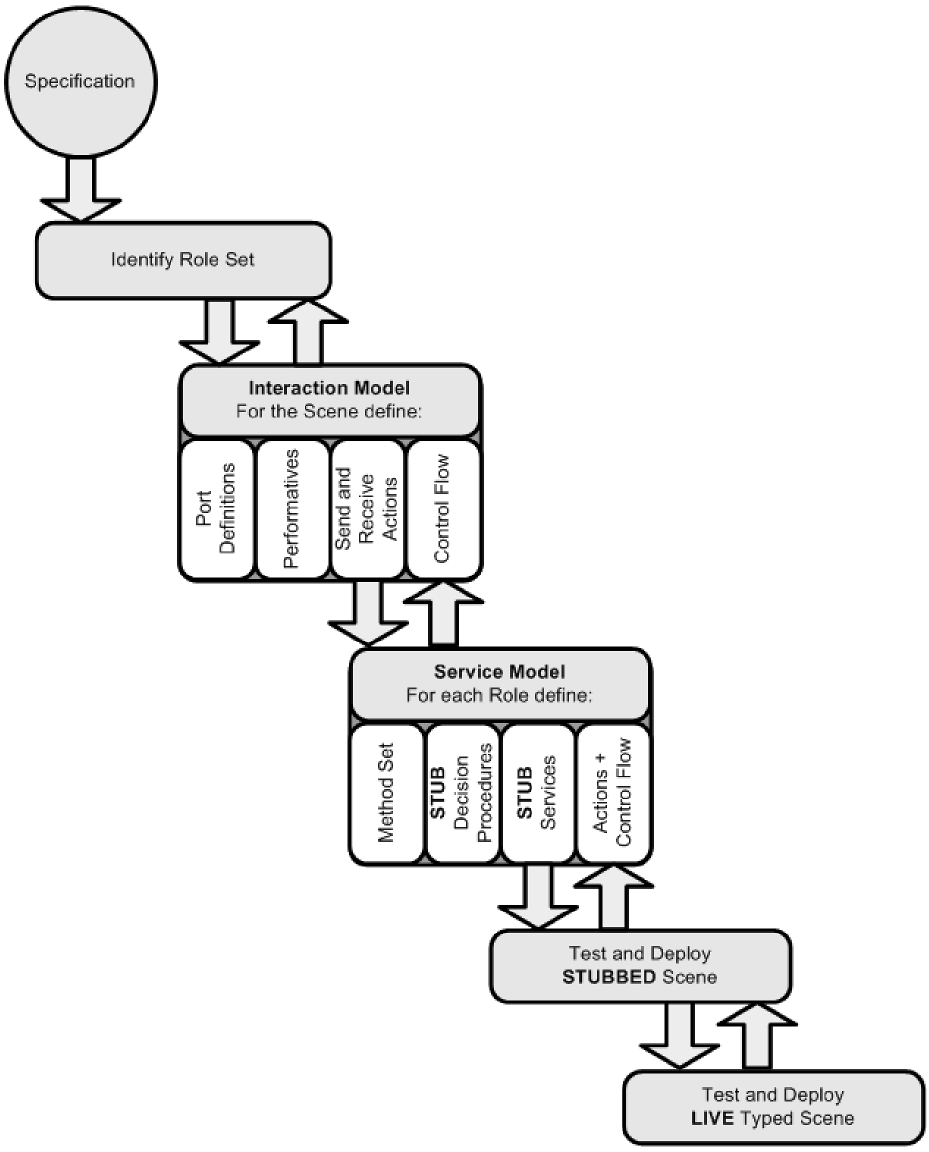


Figure 7.2: Interaction engineer methodology

The interaction engineer methodology describes the complex task of writing the protocol code to coordinate multiple, concurrent agents. The methodology is iterative and an engineer can move between phases until a working system is built that meets the original specification. The methodology is illustrated by Figure 7.2 and detailed below:

- **Identify Role Set:** Role types (as discussed in Section 5.2.2) specify a pattern of computational behaviour which an agent can adopt. The first task an engineer must perform is to break the initial specification into a number of agent role types which represent a Scene. This could be a single agent role, or multiple roles which are expected to interact as part of a multiagent system.

- **Interaction Model:** The interaction model captures any dataflow information associated with a Scene and the pattern of interaction between multiple, concurrent agent roles. The first property defines whether the Scene has any input or output port definitions, allowing it to be treated as a composable object through the dataflow layer. If the specification has been broken down into multiple role types an engineer must begin to define the performative (message type) set and specify the pattern of interaction (sending and receiving) between the agent roles within the Scene. The sending and receiving actions can (if necessary) be sugared with control flow (then, or, par etc.).

- **Service Model:** The service model fleshes out the role type definitions, allowing engineers to make use of the remainder of the action set, building around the interaction model defined by the previous stage. For each agent role the service model specifies how that role is broken down into a group of methods, making use of the remainder of the action set and control flow operators. Interaction engineers must consider how agents connect to their internal reasoning layer through decision procedure calls, specifics such as decision procedure names, input and output parameters, any faults etc. form an API skeleton which an agent engineer can then implement to achieve personalised behaviour. If the role makes use of any external services these must also be specified.

- **Test and deploy stubbed scene:** As a Scene defines an executable specification, this pattern of interaction can be tested by simply allowing agents to invoke stubbed services (decision procedures/external services). Stubbed services can be used if the live service is not available, too costly to invoke etc. This stage highlights any problems with the interaction and service models, allowing an interaction engineer to iteratively alter and test the Scene.

- **Test and deploy live scene:** Once the iterative process has terminated and the interaction engineer is confident that both the interaction model and service models are correct, live decision procedures and external services can be plugged into the Scene.

7.1.2 Experiment Engineer

As discussed by Section 3.4 scientific workflows tend to have an execution model which emphasises dataflow. Scientists are generally not skilled programmers and have no interest in the low level specifics of service composition. A scientist can therefore approach the MASC language from its most abstract level, adopting the role of an experiment engineer. An experiment engineer can use Scenes of computation (designed by an interaction engineer) as abstract objects, treating them as parameterisable black boxes of computation. As discussed in Section 5.2.6 the MASC dataflow layer allows an engineer to construct a computational experiment by providing a mapping from sources to sinks.

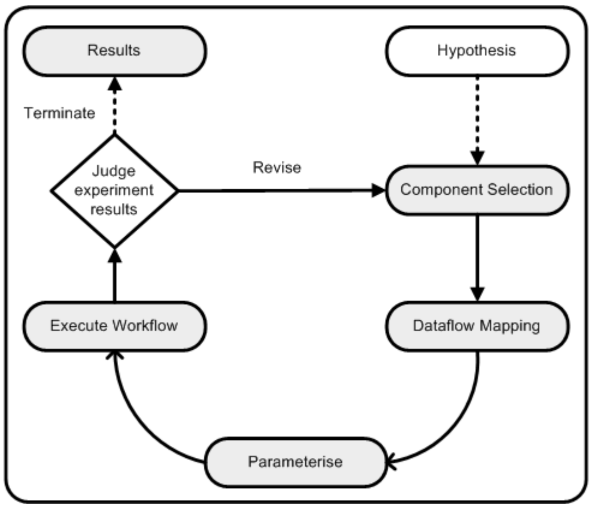


Figure 7.3: Experiment engineer methodology

Figure 7.3 illustrates the experiment engineer methodology. Scientists approach the system with a hypothesis and aim to construct a high level experiment in order to falsify that hypothesis. Components (such as Scenes (defined by the MASC language), files, services etc.) are treated as abstract problem solving components, scientists can then parameterise these components and provide a dataflow mapping which wires these components together, forming an executable experiment specification. This is an iterative, exploratory design process, steered by a hypothesis. The refinement process terminates when a suitable combination of workflow components and parameters falsify the original hypothesis.

7.1.3 Agent Engineer

As discussed in Section 2.3 a key property of agents over existing software entities is that they are able to perform autonomous action in an environment in order to meet their design objectives. In order to achieve this within the MASC framework one must adopt the role of an *agent engineer*. Agent engineers are concerned with designing customised, intelligent agents that adopt a role from a predefined Scene definition (defined by an interaction engineer). To achieve this specialised behaviour an agent engineer must implement the decision procedure set for a given role type. This behaviour can be as simple or complex as the agent engineer specifies and allows agents to have a personalised strategy (which is hidden to other agents) within the interaction model which the Scene defines.

7.2 Solving the Motivating Workflow Scenarios

This section brings together all of the concepts discussed so far by the thesis and demonstrates through example how the MASC language and coordination-oriented programming methodology (where applicable) can solve the original set of motivating workflow scenarios presented by Sections 3.2, 3.3 and 4.1. Each subsection will present in turn a workflow scenario and demonstrate how the various stages of the coordination-oriented programming methodology builds up a working protocol to solve the original specification. Workflow scenarios are ordered by complexity, starting with the most basic (utilising simple language features) to complex coordination (utilising the full language). This process includes a full implementation using the abstract MASC language and notation described by Section 5.2, lines of interest are marked by a number (not necessarily in order) and discussed by the corresponding text. Each workflow has been fully implemented and executed with live services and data on our agent-based web services composition framework. The XML representation (used as input) make up the relevant appendix.

It is important to note that with any language the definitions provided are only one possible way of solving the original specification, others exist and are equally as valid. For simplicity type information is left out of the MASC definitions, apart from special cases where `$var_name:a` is a variable of type agent name, `$var_name:alist` is a list of agent names and `$var_name:r` is variable of type role name.

7.2.1 Solving Scenario 1: Batch Processing

This Section demonstrates how the AstroGrid batch processing scenario discussed in detail in Section 3.2 can be implemented using the MASC language. This scenario is the simplest of the workflow scenarios addressed by this Chapter and takes advantage of the basic features of the MASC language, such as method definitions, sending and receiving, the sequence operator (then) and external service invocations. The corresponding XML definition (used as input) is contained in Appendix B.

```
%rsm{
  method main() =
    waitfor
1       (request($ra, $dec) <= agent($user_config:a, %user))
    timeout(e)
2      then invoke retrieve($ra, $dec, $user_config:a)
6      then invoke main()

 method retrieve($ra, $dec, $user_config:a) =
3    $images = service(def(!wfs), $ra, $dec)
     then $s_extrator = service(def(!s_extractor), $images)
     then $xmatched = service(def(!xmatcher), $sextrator)
4    then invoke redshift($xmatched, $user_config:a)

  method redshift($xmatched, $user_config) =
      $hyperz = service(def(!xmatcher), $xmatched)
5     then response($hyperz) => agent($user_config:a, %user)}.
```

Figure 7.4: Batch processing scenario - rsm role definition

Figure 7.4 is a protocol definition demonstrating one possible solution to the batch processing scenario. The interaction is broken up into two roles: `rsm` and `user`. The `rsm` role defines the pattern of interaction necessary to perform the series of service invocations needed to calculate the redshift for a given area of sky. This area of sky is supplied by the `user` agent which simply waits for a human user's input and forwards it to an agent who has assumed the role of `rsm`. Throughout the remainder of this section we will refer to Figure 7.4 by line number.

By default agents begin execution from the `main` method. Once instantiated the `rsm` agent begins its execution cycle by entering a `waitfor` loop, here it waits for a message of performative type `request` (line 1 of Figure 7.4) to be received from any agent who has adopted the role of `user`. `Waitfor` loops continue to execute until successful, or until the timeout value (an integer) is reached. In this case, the timeout value is not set, so the agent will continue to loop until the required message is received. Once a message is received (conforming to the template defined in line 1 of Figure 7.4), the area of sky requested by the user is bound to the agent's local variables: `$ra` and `$dec`. For correspondence later in the protocol the name of the `user` agent which sent the message is bound to the local variable `$user_config`. At this point the `waitfor` loop will terminate as all actions contained within the body of the loop have been successful. As the left hand side of the sequence operator (then) has been successful the right hand side is executed, invoking the `retrieve` method (line 2 of Figure 7.4) using the newly bound variables: `$ra, $dec` and `$user_config` as input parameters.

Control then passes to the `retrieve` method which initially makes a web service invocation (line 3 of Figure 7.4) in order to obtain images from each of the five wavebands necessary to compute the redshift. The web service contacts the Wide Field Survey archive (WFS) using the hard-coded service description contained in the constant: `!wfs` and the variables: `$ra, $dec` as input parameters. The result of the invocation is stored in the newly created variable: `$images`. The raw data is filtered through two analysis tools, accessed through service invocations. The first is `s_extractor`, this tool extracts from each of the images the positions of objects of interest, storing them in a Virtual Observatory table for each of the bandwidths: `$s_extractor`. The s_extractor service definition is stored in the constant: `!s_extractor`. The output of this service invocation (group of tables) is then passed through a cross matching tool which extracts all of the objects which overlap in each of the five bandwidths, storing them in just one Virtual Observatory table: `$xmatched`. If successful the `redshift` maker method is invoked (line 4 of Figure 7.4) using the variables: `$xmatched` and `$user_config` as input parameters. The redshift is calculated by invoking the service specified in the constant: `!hyperz`, using the cross matched results as input. Finally the results of this computation are sent back to the specific agent who initially requested the calculation (line 5 of Figure 7.4), as the first parameter contains the agent name: `user_config` and the second a role type: `%user`. Once control has passed back to the `main` method, the agent restarts itself by making a recursive call to `main` (line 6), here it waits for another `user` agent's request.

7.2.2 Solving Scenario 2: Knowledge Acquisition

In order to implement the knowledge acquisition scenario (discussed in detail in Section 3.3) an engineer must take advantage of more complex features of the MASC language. These features include: iterative and recursive agent definitions, the choice operator (or), agent decision procedures, web service invocations and message passing between multiple concurrent agents. The corresponding XML definition (used as input) is contained in Appendix C.

To solve the knowledge acquisition scenario a Scene containing four roles has been defined: a `user` role, `bcg` role, `extraction` role and `endpoint` role. To briefly summarise the solution, an agent representing a user's interests receives a request to conduct an experiment exploring properties of brightest cluster galaxies, this request (along with possible coordinates) are forwarded to a `bcg` agent which understands how to perform coordination to solve this type of problem. The `bcg` agent performs a registry search of databases containing information about clusters of galaxies, sending these to an agent which has adopted the `extraction` role. The `extraction` agent recursively traverses this list, sending each request to an `endpoint` agent who is responsible for performing the service invocations and sending the results back to the `extraction` agent. Once all the results are received they are stored in the AstroGrid myspace facility and the resulting URL is sent back to the originating `bcg` agent. This process is repeated for databases containing information of optical, near infrared and radio sources. The combined data sets are run through a series of web service invocations which together compute properties of brightest cluster galaxies, the results are returned to the `user` agent and forwarded to the scientist the agent represents.

Figure 7.5 illustrates the interaction model between the agent roles in the Scene and

Figure 7.1 summaries the service model. Based on the interaction and service models we have defined protocol definitions for the most complex roles: `bcg` (Figure 7.7) and `extraction` (Figure 7.6) which implement the motivating workflow scenario. These definitions and the features they make use of will now be discussed in detail.

Scene: Brightest Cluster Galaxies	
Role	**bcg**
External Services	$galaxies = service(def(!registry), !galaxies)
	$working_set = service(def(!stats1), $galaxy_data, $extra_data)
	$top_attributes = service(def(!stats2), $working_set)
	$result = service(def(!stats), $parameter_list)
Role	**extraction**
Decision Procedures	($head, $tail) = ExtractNext($qlist) fault emptylist
	Store($name, $res)
	$data = Retrieve()
External Services	$resulturl = service(def(!myspace), $data)

Table 7.1: Knowledge acquisition scenario Service Model

Once agents have received a copy of the protocol adopting the necessary roles the execution cycle can begin. In this instance the protocol execution begins with the `bcg` agent, it enters a waitfor loop, expecting to receive a message of performative type: `begin` from an agent which has adopted the `user` role. Once received, a service invocation is made to the AstroGrid registry (line 1 of Figure 7.6) in order to look up catalogues containing data about clusters of galaxies. The list returned from this service invocation is stored in the variable: `$galaxies` and sent to any agent which has subscribed to the `extraction` role (line 2 of Figure 7.6). Once sent an invocation is made to the `wait` method using the name of the originating `user` agent: `user_config` as input.

The corresponding receive on the `extraction` agent is specified on line 1 of Figure 7.7. Once a message which matches the performative: `extract` and message specification is received the contents of the message are bound to the variable: `$qlist`, this variable is the output from the AstroGrid registry lookup and contains a list of service calls which need to be made in order to obtain the required data from the bcg calculation. An invocation is then made to the `eloop` method, using the query list: `$qlist` and name of the originating agent: `$bcg:a` as parameters.

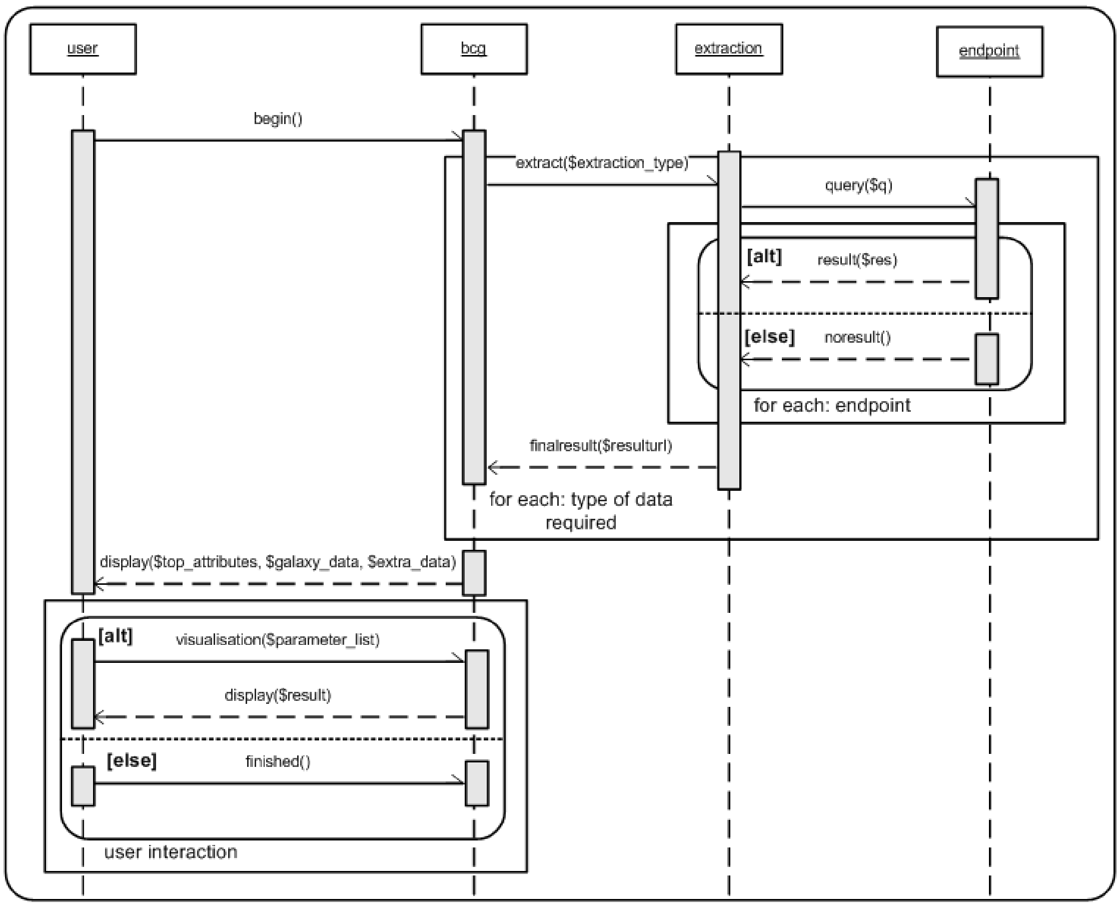


Figure 7.5: Knowledge acquisition scenario - Interaction Model

```
%bcg{
  method main() =
    waitfor
      (begin() <= agent($user_config:a, %user)
1     then $galaxies = service(def(!registry), !galaxies)
2     then extract($galaxies) => agent(_, %extraction)
      then invoke wait($user_config:a))
    timeout(e)

  method wait($user_config:a) =
    waitfor
3     (finalresult($galaxy_data) <= agent(_, %extraction))
    timeout(e)
4     then $extra = service(def(!registry), !radio)
5     then extract($extra) => agent(_, %extraction)
    waitfor
6     (finalresult($extra_data) <= agent(_, %extraction)
      then invoke bcg($galaxy_data, $extra_data, $user_config:a))
    timeout(e)

  method bcg($galaxy_data, $extra_data) =
7   $working_set = service(def(!stats1), $galaxy_data, $extra_data)
8   then $top_attributes = service(def(!stats2), $working_set)
9   then display($top_attributes, $galaxy_data, $extra_data)
                                                 => agent($user_config:a, %user)
    then invoke userinteraction($user_config:a)

  method userinteraction($user_config:a) =
    waitfor
10    ((visualisation($parameter_list) <= agent(_, %user)
      then $result = service(def(!stats), $parameter_list)
      then display ($result) => agent($user_config:a, %user))
11      or (finished() <= agent(_, %user)
           then invoke main()))
    timeout(e)}
```

Figure 7.6: Knowledge acquisition scenario - bcg role definition

```
%extraction{
  method main() =
   waitfor
1     (extract($qlist) <= agent($bcg:a, %bcg)
     then invoke eloop($qlist, $bcg:a)
     then invoke main())
   timeout (e)

  method eloop($qlist, $bcg:a) =
2  (($head, $tail) = ExtractNext($qlist) fault emptylist
   then query($head) => agent(_, %endpoint)
3  then invoke eloop($tail, $bcg:a))
4    or (invoke ewait($bcg:a))

  method ewait($bcg:a) =
    waitfor
5    (((result($res) <= agent($name, %endpoint)
     then Store($name, $res)
     then invoke ewait($bcg:a)))
6     or (noresult() <= agent($name, %endpoint)
      then invoke ewait($bcg:a)))
7   timeout (invoke end($bcg:a)

   method end($bcg:a) =
     $data = Retrieve()
     then $resulturl = service(def(!myspace), $data)
8    finalresult($resulturl) => agent($bcg:a, %bcg)}}.
```

Figure 7.7: Knowledge acquisition scenario - extraction role definition

The `eloop` method makes use of three language features not demonstrated so far by this Chapter: recursive function calls, basic agent internal reasoning and the choice operator. This method recursively traverses the list of queries, constructing and sending each unique query to the appropriate service. The method begins by invoking the `HeadTail` decision procedure (line 2 of Figure 7.7), which removes the item at the front of the list: `$head`, and stores the remainder of the list in: `$tail`. The `HeadTail` decision procedure is a call to the agent's internal decision logic that an agent engineer has written, deployed as a reasoning web service and associated to bcg agent before execution began. It was associated by supplying the WSDL address of the web service which implements all of the decision procedures contained in the service model for the `extraction` role type.

If the `HeadTail` decision procedure is successfully executed then the current query: `$head` is sent to any agent which has subscribed to the `endpoint` role. If sending is successful a recursive invocation is made to the `eloop` method (line 3 of Figure 7.7) using the tail of the list as the first input parameter. The recursion will terminate if any one of the sequential statements (constructed using the `then` operator) fail. A valid termination would be the result of the `HeadTail` decision procedure raising the fault `emptylist`, indicating there are no more items to process. Once a statement fails or an exception is raised the `or` branch of the protocol is executed (line 4 of Figure 7.7), which in this case makes an invocation to the `ewait` method.

Once all the queries have been sent out, the `endpoint` agents will process them by invoking the services and sending the results back. It is the job of the `ewait` method to collect and send back these responses to the original agent, bound to the `$bcg:a` variable. Inside this method the agent is waiting for two types of message indicated by the performative type and message structure, either: `result` or `noresult`, this is achieved by separating the receives with an `or` operator. If a query has been successful a `result` message is received (line 5 of Figure 7.7) containing the processed query: `$res`. This query is then stored locally to the agent by making an invocation to the `Store` decision procedure, providing the name of the agent: `$name` and the processed query: `$res`. A recursive invocation is then made to the `ewait` method. The choice operator here allows the agent to listen for several kinds of message, in this case the other option is to receive a `noresult` message (line 6 of Figure 7.7) indicating that an agent has not been able to process the query. Within our protocol definition, failure is simply ignored and the agent makes a recursive invocation to the `ewait` method.

The loop will terminate when the agent has been waiting for messages over the stated `timeout` period (line 7 of Figure 7.7), when the loop times out an invocation to the `end` method is made.

The terminating method invokes the `Retrieve` decision procedure, storing the accumulated processed data into the `$data` variable. The result of the extractions are then sent to the AstroGrid storage facility (line 8 of Figure 7.7) through an external service invocation, the details of which are stored in the constant: `!myspace`. The result of this invocation is a URL, pointing to where the published results reside: `$resulturl`. This URL is sent back to the agent which made the original request for extraction: `$bcg:a` for further analysis. The agent restarts itself by making a call to the `main` method.

The `bcg` agent is expecting to receive (line 3 of Figure 7.6) a message of performative type `result`. Once received the entire process is repeated this time for data sources which are classified as catalogues of optical, near-infrared and radio sources and which, therefore, might include relevant observations of BCGs (lines 4,5,6 of Figure 7.6). The results from both of these extraction processes: `$galaxy_data` and `$extra_data` are passed in as parameters to the `bcg` method through an invocation. The `bcg` method passes the results from both extraction processes through a series of web service invocations. The first works out which galaxies in the galaxy catalogue data are the BCGs in each of the host clusters (line 7 of Figure 7.6) generating a combined set of all the data known about each Cluster/BCG pair. The second is an invocation to a statistics algorithm (line 8 of Figure 7.6), which seeks the twenty attributes with the highest information content on the deposited data. The output of these service invocations are forwarded to an agent which represents a human scientist (line 9 of Figure 7.6), stored in the variable `$user_config:a`. The scientist can use their expertise and give judgement about how the workflow should progress. It is important to keep the scientist in the loop, in this case the astronomer must step back and look at the data, the visualisation tool displays a set of scatter plots which are judged as possibly worthy of further investigation.

Once the data has been sent an invocation to the `userinteraction` method is made where the agent is waiting for input from the user scientist. Input from the scientist can take the form of two performatives: `visualisation` (line 10 of Figure 7.6) or `finished` (line 11 of Figure 7.6), separated using the `or` operator. The `bcg` agent responds to the user's commands by invoking further statistics on the results in accordance to the supplied parameters stored in the variable: `$parameter_list`. User interaction terminates when the `finished` performative is received.

7.2.3 Solving Scenario 3: Runtime Coordination

This Section discusses the most complex of our motivating workflow scenarios, taken from the Large Synoptic Survey Telescope (LSST) [56]. This workflow scenario is discussed in detail in Chapter 4 acting as a pivotal point in this thesis. This Section will practically demonstrate how the MASC language can solve a scenario requiring flexible, runtime composition of services and data in an inherently distributed, peer-to-peer environment. The implementation makes use of some advanced features of the language, allowing components to be composed at runtime and not (as previously demonstrated) be hard-coded into the protocol. The features demonstrated include: port reading and writing, creating or locating agents on the fly, message passing based on agent type or role type at runtime, and service invocation at runtime. We also demonstrate how the dataflow layer can be used to compose experiments at a higher level of abstraction, for use in the scientific community. The corresponding XML definition (used as input) is contained in Appendix D.

To solve the time-domain astronomy workflow scenario we have divided the Scene into four distinct roles: `classification` (Figure 7.9), `contractnet` (Figure 7.10), `observatory` (Figure 7.11) and `extraction` (Figure 7.12). The interaction model is illustrated by Figure 7.8 and the service model by Figure 7.2.

To briefly summarise the implementation, a `classification` agent is attempting to locally classify a list containing pointers to objects which cannot be classified by the automated algorithms discussed in Section 4.1.1. If at any time the agent cannot classify an object locally help needs to be obtained from agents running at distributed observatories. The first port of call is a request to an agent which has adopted the `contractnet` role, supplying as parameters to the message: a list of suitable agents: `$potential_agents` (obtained from a registry lookup) and a proposal defining the terms of agreement: `$proposal`. The `contractnet` agent is responsible for executing the contractnet protocol [50], requesting participation from each agent in the list `$potential_agents`. This list of agents is the result of a registry lookup at runtime, therefore the agents available are very much dependent on the facilities of the observatory, current work schedule etc. Once the call for participation has finished the `contractnet` agent returns a list of agents working at observatories who returned `propose` to the protocol. The list of open proposals is then evaluated locally (according to some internal constraint defined by a decision procedure invocation) by the `classification` agent, generating a list of rejected agents: `$reject` and a single suitable agent: `$accept`. An `accept-proposal` message is sent to the selected agent. The agent working on behalf of the observatory breaks down the proposal, farming out the computational job to a number of `extraction` agents, the number and location of which are decided at runtime. Each `extraction` agent receives a section of the original proposal containing a list of services, with parameter settings etc. to invoke. The `extraction` agent then calls each service in turn, setting the input and output parameters on the fly, returning the results to the agent working on behalf of the observatory. If the terms of the proposal have been fulfilled successfully an `inform-result` message is sent to the originating `classification` agent. The data received from the distributed observatory: `$opinion` is used to generate a `$combined_opinion`, informing a human scientist if anything unusual has occurred, requiring followup observations etc. The agent then continues to classify the remaining objects. In parallel to this task taking place, the agents who were unsuccessful in the contractnet proposal bid are rejected by the `reject-proposal` message. However, if the contractnet proposal has been unsuccessful another attempt must be made (by executing the process described above again) to find suitable data from distributed observatories. The contractnet proposal can fail in three ways, firstly if the list: `$open_proposals` is empty (throwing the `noproposals` fault), secondly if the `classification` agent has been waiting too long and `timeout` is reached and finally if the observatory selected cannot fulfill the terms of the contract sending an `inform-failure` message.

As most of the basic features have already been discussed by the previous workflow scenarios this explanation of the implementation will only focus on the more advanced features of the language. The implementation makes simple use of the dataflow layer, the `objectclassification` Scene has two ports, an inport: `lsst:lsst_in1` (line 1 of Figure 7.12) and an outport: `lsst:out1` (line 2 of Figure 7.12). As the mapping demonstrates, an outport from the automated processing software (discussed in Section 4.1.1) is connected to the inport of the `objectclassification` Scene (line 3 of Figure 7.12) and the outport from the `objectclassification` Scene is connected to an inport of a Scene handling user interaction (line 4 of Figure 7.12. The first use of this mapping is made through a `portread` operation (line 1 of Figure 7.9). As the automated classification software runs it outputs any data that cannot be classified to

Scene: Runtime coordination	
Inports	lsst_in1
Outports	lsst_out1
Role	**classification**
Decision Procedures	$result = StatTest($unknown)
	UpdateKnowledge($result)
	QueryKnowledge($unknown) fault cannotclassify
	$proposal = GenerateProposal($unknown)
	($accept, $reject) = Evaluate($open_proposals) fault noproposals
	($head, $tail) = HeadTail($reject) fault emptylist
	AgentCheck($accept, $observatory) fault wrongagent
	$combined_opinion = GenerateOpinion($opinion)
External Services	$potential_agents = service(def(!registry), $unknown)
Role	**contractnet**
Decision Procedures	$c_id = GenerateID()
	($head, $tail) = HeadTail($potential_agents) fault emptylist
	($name, $role) = NameRole($head)
	StoreProposal($agent, $id, $c_id) wrongcid
	StoreRefusal($agent, $id, $c_id) wrongcid
	$open_proposals = RetrieveProposals()
External Services	N/A
Role	**observatory**
Decision Procedures	ConsiderProposal($proposal) fault proposalrefused
	AgentCheck($initiator, $init) fault wrongagent
	$proposal_sections = DivideProposal($proposal)
	($head, $tail) = HeadTail($proposal_sections) fault emptylist
	Finished(), Remove()
	Store($data)
	$opinion = ExtractData()
External Services	N/A
Role	**extraction**
Decision Procedures	($head, $tail) = HeadTail($proposal_section) fault emptylist
	$service_def = ExtractService($head)
	$service_input = ExtractInput($head)
	$service_output = ExtractOutput($head)
	Store($service_output)
	$data = Retrieve()
External Services	$service_output = service(def($service_def), $service_input)

Table 7.2: Runtime coordination scenario service model

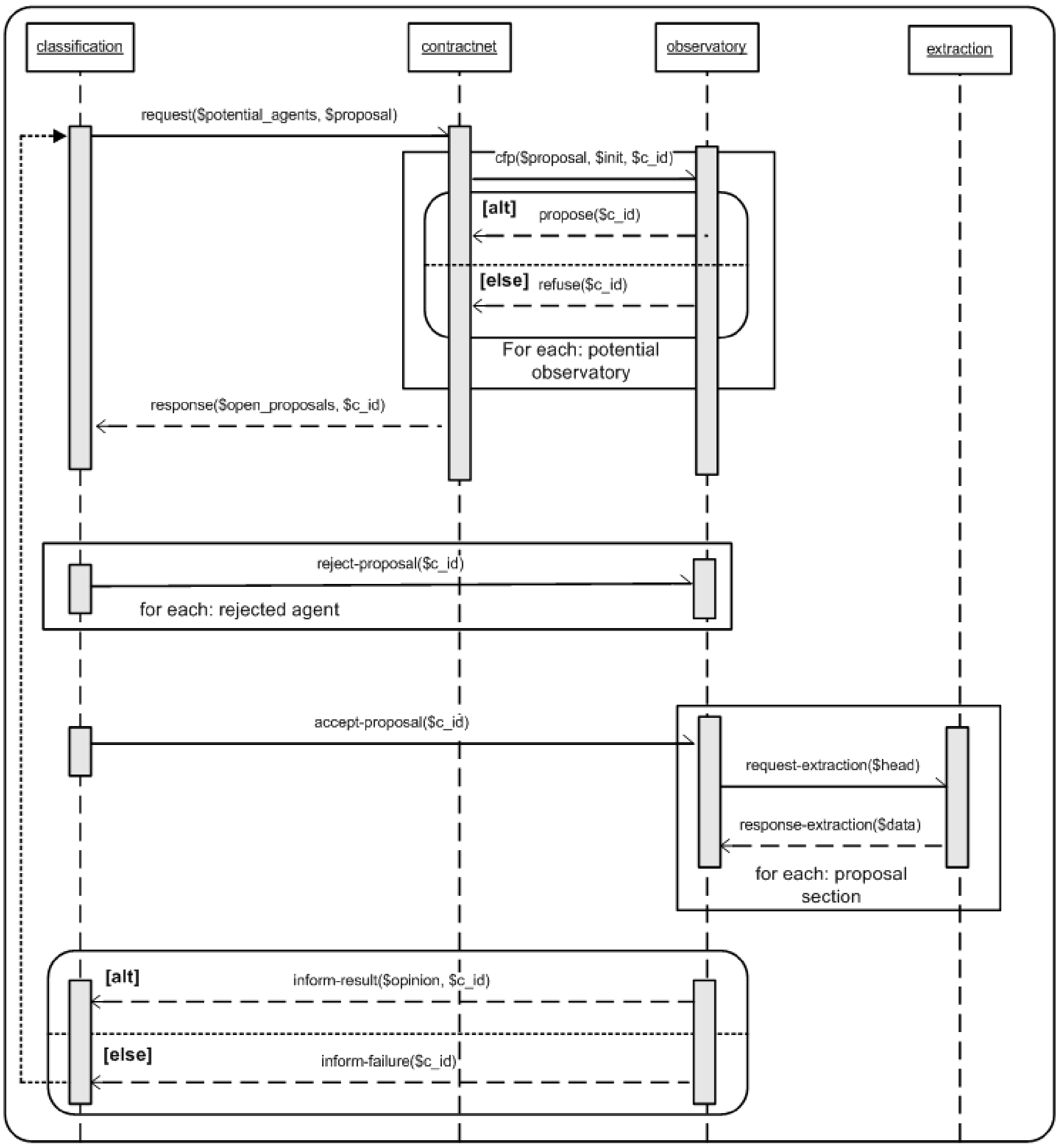

classification
contractnet
observatory
extraction
request($potential_agents, $proposal)
cfp($proposal, $init, $c_id)
[alt]
propose($c_id)
[else]
refuse($c_id)
For each: potential observatory
response($open_proposals, $c_id)
reject-proposal($c_id)
for each: rejected agent
accept-proposal($c_id)
request-extraction($head)
response-extraction($data)
for each: proposal section
[alt]
inform-result($opinion, $c_id)
[else]
inform-failure($c_id)

Figure 7.8: Runtime coordination scenario - Interaction Model

```
protocol(objectclassification, {
scene{lsst, {
%classification{
   method main() =
1    $unknown = portread(lsst:lsst_in1)
     then $result = StatTest($unknown)
     then UpdateKnowledge($result)
     then (QueryKnowledge($unknown) fault cannotclassify
     then invoke main()
   or invoke contractnetsend($unknown)) or e)

  method contractnetsend($unknown) =
2  $potential_agents = service(def(!registry), $unknown)
   then $proposal = GenerateProposal($unknown)
   then request($potential_agents, $proposal) => agent(_, %contractnet)
   then waitfor
     (response($open_proposals:alist, $c_id) <= agent(_, %contractnet)
3     then ($accept:a, $reject:alist) = Evaluate($open_proposals:alist) fault noproposals
      then invoke contractreject(($reject:alist, $c_id, $accept:a, $unknown))
   timeout(invoke contractnetsend($unknown))

  method contractreject($reject:alist, $c_id, $accept:a, $unknown) =
   (($head:a, $tail:alist) = HeadTail($reject:alist) fault emptylist
    then reject-proposal($c_id) => agent($head:a, _)
    then invoke contractreject($tail:alist, $c_id, $accept:a, $unknown))
  or (invoke contractaccept($accept:a, $unknown, $c_id))

  method contractaccept($accept:a, $unknown, $c_id) =
    accept-proposal($c_id) => agent($accept:a, _)
    then waitfor
      (inform-result($opinion, $c_id) <= agent($observatory:a, _)
       then AgentCheck($accept:a, observatory:a) fault wrongagent
       then $combined_opinion = GenerateOpinion($opinion)
4      then portwrite(lsst:lsst_out1, $combined_opinion)
       then invoke main())
      or (inform-failure($c_id) <= agent($observatory:a, _)
5      then invoke contractnetsend($unknown))
    timeout(e)},
```

Figure 7.9: Runtime coordination scenario - classification role definition

```
%contractnet{
  method main() =
    waitfor
      (request($potential_agents, $proposal) <= agent($initiator:a, _)
      then $c_id = GenerateID()
      then invoke cfp($potential_agents, $proposal, $initiator:a, $c_id)
      then invoke main())
    timeout(e)

  method cfp($potential_agents, $proposal, $initiator:a, $c_id) =
    (($head, $tail) = HeadTail($potential_agents) fault emptylist
1   ($name:a, $role:r) = NameRole($head)
    then cfp($proposal, $initiator:a, c_id) => agent($name:a, $role:r)
    then invoke cfp($tail, $proposal, $initiator:a, $c_id))
    or (invoke receiveproposals($initiator:a, $c_id))

  method receiveproposals($initiator:a, $c_id) =
    waitfor
      ((propose($id) <= agent($agent:a, _)
      then StoreProposal($agent:a, $id, $c_id) fault wrongcid
      then invoke receiveproposals($initiator:a))
      or (refuse($id) <= agent($agent:a, _)
      then StoreRefusal($agent:a, $id, $c_id) fault wrongcid
      then invoke receiveproposals($initiator:a)))
    timeout($open_proposals:alist = RetrieveProposals()
2            then response($open_proposals:alist, $c_id) => agent($initiator:a, _))},
```

Figure 7.10: Runtime coordination scenario - ContractNet role definition

```
%observatory{
  method main() =
    waitfor
      (cfp($proposal, $initiator:a, $c_id) <= agent($contractnet:a, %contractnet))
    timeout(e)
1     (then ConsiderProposal($proposal) fault proposalrefused
      then propose($c_id) => agent($contractnet:a, %contractnet)
      then invoke waitfordecision($initiator:a, $proposal))
      or (refuse($c_id) => agent($contractnet:a, %contractnet)
      then invoke main())



  method waitfordecision($initiator:a, $proposal) =
    waitfor
      ((accept-proposal($c_id) <= agent($init:a, _)
      then AgentCheck($initiator:a, init:a) fault wrongagent
2     $proposal_sections = DivideProposal($proposal)
      then invoke extract($proposal_sections, $initiator:a, $c_id))
      or (reject-proposal($c_id) <= agent($init:a, _)
        then AgentCheck($initiator:a, init:a) fault wrongagent
        then invoke main()))
    timeout(e)

  method extract($proposal_sections, $initiator:a, $c_id) =
    (($head, $tail) = HeadTail($proposal_sections) fault emptylist
    then request-extraction($head) => agent(_, %extraction)
    then invoke extract($tail, $initiator:a, $c_id))
    or (invoke wait($initiator:a, $c_id))

  method wait($initiator:a, $c_id) =
    (Finished()
    then waitfor
      (response-extraction($data) <= agent(_, %extraction)
      then Store($data)
      then Remove()
      then invoke wait($initiator:a, $c_id))
    timeout(inform-failure($c_id) => agent($initiator:a, _))
    or (invoke finish($initiator:a, $c_id))

  method finish($initiator:a, $c_id) =
    $opinion = ExtractData()
3   then inform-result($opinion, $c_id) => agent($initiator:a, _)
    then invoke main()},
```

Figure 7.11: Runtime coordination scenario - observatory role definition

```
%extraction{
  method main() =
    waitfor
      (request-extraction($proposal_section) <= agent($coordinator:a, _)
      then invoke retrieve($proposal_section, $coordinator:a)
      then invoke main())
    timeout(e)

  method retrieve($proposal_section, $coordinator:a) =
5   (($head, $tail) = HeadTail($proposal_section) fault emptylist
6   then $service_def = ExtractService($head)
7   then $service_input = ExtractInput($head)
8   then $service_output = ExtractOutput($head)
9   then $service_output = service(def($service_def), $service_input)
    then Store($service_output)
    then invoke retrieve($tail, $coordinator:a))
    or ($data = Retrieve()
10  then response-extraction($data) => agent($coordinator:a, _))}},

1   {inport(lsst:lsst_in1, true)},
2   {outport(lsst:lsst_out1)},
3   {link outport(automated:auto_out1) -> inport(lsst:lsst_in1),
4    link outport(lsst:lsst_out1) -> inport(user:user_in1)}).
```

Figure 7.12: Runtime coordination scenario - extraction role definition and dataflow mapping

the outport: `automated:auto_out1`, which we have just discussed has been mapped to the inport: `lsst:lsst_in1`. Therefore when the `portread` operation is invoked it removes the first item that cannot be classified, storing it in the variable: `$unknown`. If the `lsst` agent cannot classify the item: `$unknown` locally observatories need to be located at runtime gathering evidence on whether this unknown object is potentially a new species of object, or simply some kind of equipment failure etc. The location of suitable agents is made by contacting a registry (line 2 of Figure 7.9) through a service invocation, using the unknown object: `$unknown` as input to the invocation. Based on the object type, coordinates etc. the registry lookup returns a list of agents formatted as name, role pairs, storing this list in the newly created variable: `$potential_agents`.

A proposal of work based on: `$unknown` is generated through a decision procedure invocation, this along with `$potential_agents` is sent to any agent which has subscribed to the `contractnet` role. Once received by the `contractnet` agent the list is recursively traversed, extracting the name: `$name:a` and role `$role:r` of the agent to issue the proposal to (line 1 of Figure 7.10). The name and role of the agents to issue the proposal to cannot be hard-coded into the protocol as this list is decided purely at runtime through a registry lookup. The registry lookup is itself dependent firstly on the unknown object (which the `lsst` agent cannot predict) and secondly on external influences such as: network conditions, current load of agents working at distributed observatories. Once the call for participation has been sent to every agent in the list: `$potential_agents` it is then up to the agent working on behalf of the observatory to autonomously decide whether it is willing to fulfill the terms of the proposal or not. As an example we have implemented such a role: `%observatory`. Once the proposal is received the `ConsiderProposal` decision procedure (line 1 of Figure 7.11) is invoked. Based on some internal constraint (programmed by an agent engineer) that is not visible to the rest of the multiagent system the agent will either issue: `propose` or `refuse` (if the `proposalrefused` fault is thrown). Once all agents working on behalf of an observatory have made an autonomous decision the list of agents that returned `propose` is sent back to the `lsst` agent (line 2 of Figure 7.10).

Once the `lsst` agent has received this list of open proposals it must then itself autonomously decide which agents to reject and which single agent to accept. This is decided through a decision procedure invocation: `Evaluate` (line 3 of Figure 7.9) programmed by an agent engineer, the output is based on the quality of participants, costs involved, how quickly the observatory could fulfill the terms of the proposal etc. Once decided, all agents in the list: `$reject:alist` are issued the `reject-proposal` message and the chosen agent: `$accept` is issued the `accept-proposal` message. The agent working on behalf of the observatory which has been successful in the contract-net bid then divides the proposal into a number of sections (line 2 of Figure 7.11) through a decision procedure invocation: `DivideProposal`. The number of sections that the proposal is divided into is again purely based on a runtime decision, depending how much work is involved with the proposal, current work schedule etc. Once divided each proposal section is issued to an agent which has subscribed to the `extraction` role.

Once an `extraction` agent (line 5 of Figure 7.12) receives the proposal section it recursively traverses it breaking it down into `$head` and `$tail`. Through a series of decision procedure invocations (lines 6-9 7.12) the agent dynmically builds an invocation model which results in an external service call. This content cannot be hard-coded into the protocol as the interaction engineer building the `extraction` agent cannot predict the series of service invocations that need to be made at design-time, it therefore must be done on-the-fly at runtime. Once the terms of the proposal section are met the results are sent back to the originating agent working on behalf of the observatory (line 10 of Figure 7.12). If the terms of the proposal have been fulfilled and the computation for all the proposal sections has been completed (line 3 of Figure 7.12) the results are sent back to the originating `lsst` agent through a `inform-result` message. Based on the updated knowledge the `lsst` agent generates a combined opinion through a decision procedure invocation: `GenerateOpinion`. The output from this invocation: `$combined_opinion` is then written to the outport `lsst:lsst_out1` through a `portwrite` operation (line 4 of Figure 7.9). As discussed earlier in this section the `lsst:lsst_out1` port is mapped to the inport: `user:user_in1`, which handles user interaction with a human scientist. For simplicity this is not discussed by this example but it should be clear how to implement this functionality using the MASC language. Once the `portwrite` is complete the `main` method is invoked, which continues to process the remaining data from the inport: `lsst:lsst_in1`.

However if the agent working on behalf of the observatory has not been able to meet the terms of the proposal an `inform-failure` message is sent to the `lsst` agent instead. If received an invocation to the `contractnetsend` method is made (line 5 of Figure 7.9) which restarts the process of locating a suitable observatory to help classify the unknown object. This process will continue to be executed until successful and each iteration will result in a different set of agents being contacted due to changing conditions from iteration to iteration.

7.3 Discussion: A Better Approach to Workflow?

Throughout this thesis we have derived a list of requirements for scientific workflow composition. It is important to note that the workflow scenarios have not been invented for the purpose of this thesis, rather this thesis and the requirements that we have derived are a consequence of analysing these workflow scenarios. It is also true that solutions to the workflow scenarios (in particular the knowledge acquisition and runtime coordination) don't readily exist. Scientists from the domain have considered similar classes of problem but the scenarios addressed by this thesis are considered *future development work*. This Section discusses how our agent-based approach to service composition fulfills the requirements of scientific workflow:

- **R1 - Rapid prototyping:** Scientists require the ability to incrementally and rapidly prototype an experiment based on a hypothesis. The MASC language allows rapid prototyping in two very different ways, the first of which is prototyping a sequence of interaction between a group of agent roles and external services. As MASC is a specification which is directly executable by a group of agents, this provides an effective mechanism for prototyping a workflow. Protocols can be used to engineer a prototype system from a scenario (like those discussed throughout this thesis) even if the exact services or interaction model (or both) are undefined at the design stage. This allows interaction engineers to focus on defining the exact pattern of interaction using stubbed services before deploying the interaction model on live services and data. This is further addressed by the coordination-oriented programming methodology discussed in Section 7.1. Secondly, experiments can be prototyped from a higher level of abstraction by adopting the role of an experiment engineer. This allows problem solving components to be treated as parameterisable black boxes of computation, wired through the dataflow layer.

- **R2 - User interaction:** The ability to interact with a user is an essential requirement of scientific workflow modelling. There are two mechanisms in the MASC language which aid this requirement. Firstly, individual agents can send messages to and receive messages from a user, these sending and receiving actions can then be wrapped around control flow operators (such as `then`, `or` etc.) to steer the execution path of an agent depending how the user reacted. Secondly user interaction can be mapped at the scene level, by binding a user to a scene's inport or binding a scene's outport to a user.

- **R3 - Workflow Reuse:** Protocols are executable specifications which can be directly enacted by a group of agents. Therefore the scene description (written by an interaction engineer) is a generic description that can be enacted by any group of agents which adopt the roles defined by the scene. This means that once a scene has been written it is fully reusable. Workflow components can also be reused from a higher level of abstraction when adopting the experiment engineer role. From this level of abstraction scenes can be wired together through the dataflow layer like any other workflow component.

- **R4 - Fault tolerant execution:** In order to keep the MASC language as lightweight as possible, no explicit fault tolerant features have been added. However, an interaction engineer can build fault tolerant protocols by taking advantage of the features included. For example, the operation set includes an `or` clause, `waitfor` loops continue to execute until successful, and `timeout` clauses specify compensation actions.

- **R5 - Levels of abstraction:** Ideally scientific workflows should be viewable and configurable from different layers of abstraction. As discussed by this Chapter the MASC language can be approached from various levels of abstraction to accommodate the differing requirements and skill sets of users. An experiment engineer can treat protocols as parameterisable black boxes of computation, wiring them together through dataflow. An interaction engineer is concerned with defining roles and specifying how those roles coordinate with one another to achieve a shared goal. Finally, an agent engineer is concerned with defining an agent's internal reasoning model by implementing a set of decision procedures. The coordination-oriented programming methodology aids each level of abstraction.

- **R6 - Legacy system integration:** Many scientific applications are considered legacy applications as they are written in older programming languages such as Fortran. Legacy applications are still widely in use and need to be integrated into existing workflow tools. Legacy applications are easily integrated into a workflow specified using the MASC language. With little engineering work these legacy applications can be wrapped up and exposed as a service, this service can then be invoked like any other piece of service-oriented architecture by the agents which act as proxies or stubs to their enactment.

- **R7 - Provenance data and R9 - Semantic markup:** The MASC language does not specify how provenance information is supplied or how services and data can be semantically marked up. This should be handled by the service providers themselves and it is up to an agent engineer to specify how individual agents utilise this extra information if available.

- **R8 - Smart component choice:** The MASC language allows agents to make decisions about which components to interact with at runtime, based on the current state of the network etc. This could be made through negotiation with other agents, variable substitution or according to the agent's local knowledge through calls to decision procedures. This concept was illustrated by the LSST runtime scenario discussed in detail in Section 7.2.3 and is discussed in more detail later in this Section.

- **R10 - Data presentation:** The Zorro framework is a prototype implementation of the concepts addressed by the MASC language. It provides the essential workflow execution engine, however with little engineering work improvements could be made to the tool, this will be discussed in more detail in the Further Work Section 8.2.

As discussed in Section 4.1.4, current service composition techniques allow *statically defined*, *pre-designed/pre-planned* workflows to be enacted by a *centralised workflow engine*. However, through our exploration of workflow scenarios we presented a counterexample of coordination which is difficult or impossible to achieve by existing service composition techniques. This process helped derive an extended list of desirable properties of a workflow language. There are a number of features of the MASC language which specifically address these requirements, allowing flexible, runtime composition of services, each of these features will now be highlighted in turn:

- **Decentralised, peer-to-peer architecture:** The MASC language is designed to be executed by a number of distributed agents, which act as peers, forming a peer-to-peer system. Before enactment of the workflow can begin each agent receives a local copy of the interaction protocol, assumes a role with that protocol and references a reasoning web service which implements the decision procedure set for the role it has assumed. Agents can therefore act as independent, self contained peers with no centralised server governing the interaction.

- **Agent reasoning through decision procedures:** MASC protocols allow the rules of interaction to be explicitly expressed, while allowing individual agents to subscribe to their own reasoning models. Protocols do not sacrifice the self interest and autonomy of individual agents, although agents follow the protocol as a script each agent can adopt their own personalised strategy within the protocol. Reasoning web services can be mapped on an individual agent basis (providing personalised behaviour) or on role type (providing generic role behaviour). It is up to the agent engineer to provide the set of methods which form this reasoning web service.

- **Agents are proxies to service invocation:** Agents add an extra level of abstraction, acting as stubs or proxies to the web services which are taking part in the workflow. This means that agents can make use of their internal reasoning (through decision procedure invocations) to make decisions at runtime when the coordination is actually taking place. This concept was illustrated by both the knowledge acquisition (Section 7.2.2) and runtime coordination scenarios (Section 7.2.3). This approach offers more than 'just coordination', provided by most web service composition frameworks and languages.

- **Variable substitution:** Most workflow languages are hardcoded specifications of execution, MASC on the other hand allows sections of the interaction to be compiled at runtime. Actions (such as sending/receiving, service invocation etc.) in the MASC language allow variable substitution. An agent, therefore can treat a protocol as a template of coordination, although the sequence of actions are defined, specific details (such as which service to invoke) can be spliced in at runtime. This allows agents to use knowledge such as the current state of the network to provide flexible service composition while the workflow is executing, instead of enacting a pre-defined, static workflow. This was demonstrated in particular by the runtime coordination scenario.

- **Recursion:** Agents can iterate over method definitions, data structures etc. recursively. This allows a more complex, expressive class of workflow to be defined.

- **Layered structure:** The MASC language fills the gap between the low level transport issues of an agent (such as network protocol etc.) and its high level rational processes. This layering removes some of the complications of designing large multiagent systems, aiding in the design process.
- **Inter-operability:** By adopting the MASC language, agents built by different organisations, using different software systems, written in different languages are able to communicate with one another in a common language with agreed semantics. The only requirement on an engineer wanting to build an agent that can coordinate is a layer of software which can translate the protocol and a mapping to a reasoning web service which implements the decision procedure set for a given role.
- **Infrastructure independent:** The interaction model always remains a layer above any implementation specific middleware or operating systems. The only time an agent needs to talk to this lower level is when it is sending and receiving messages, making calls to decision procedures or external web services. This means that as inherently unstable standards keep changing, the interaction model remains unaffected.
- **Compatibility:** The coordination mechanism defined using the MASC language is entirely external to the web services which are being coordinated. The web services themselves need no alteration or knowledge that they are even taking part in coordination. Therefore no modification of web services needs to take place and the protocol does not need to be disseminated between the web services themselves.
- **Fit in with existing architectures:** As there are several fully developed graphical service composition tools (e.g Taverna [43]), with little effort scientists can simply integrate components expressed in the MASC language into these existing frameworks. For example, adding our novel multiagent/service-oriented approach as a dataflow node in an experiment constructed using Taverna.

7.3.1 Possible Limitations of the Approach

Although we have argued that an agent-based approach to service composition has several advantages in the right domain, it is important to discuss the limitations of this approach and where this technique is not appropriate:

- **The peer-to-peer design process:** The design process for a peer-to-peer workflow is inherently more complex than a traditional centralised approach. An engineer not only has to consider ordering a set of services but also the tricky problem of message passing between multiple concurrent processes.
- **The appropriate level of complexity:** The added complexity of workflow design in a peer-to-peer system is useful with large scale distributed systems where task delegation is encouraged, but can be an added overhead for very simple workflows with just a few services. There is a trade off between task delegation and workflow complexity, an engineer needs to make a choice as to when this technique is applicable to a workflow scenario. It only makes sense to use this technique when the patterns of interaction are too complex to analyse at design-time, requiring runtime service composition.
- **Autonomy isn't always appropriate:** The agent-based approach discussed by this thesis encourages linking the protocol execution to models of agent reasoning. This agent reasoning can facilitate autonomous, runtime decision making. This technique may not always be a desirable trait as an engineer loses complete control which is taken for granted in a statically defined centralised workflow.

7.4 Chapter Conclusions

This Chapter serves as the focal point for the thesis, bringing together all of the concepts addressed so far by this research. Firstly the coordination-oriented programming methodology was proposed which serves as a guideline on how to implement workflows using our approach. Users can approach the system from various levels of abstraction, adopting the role of: an experiment engineer, interaction engineer or agent engineer depending on their aims and motivations.

Throughout this thesis scenarios have always been a driving factor, therefore it is logical to perform the evaluation by case-study. Our agent-based approach to service composition (using the coordination-oriented programming methodology) was applied to each of the motivating workflow scenarios, taken from the live Grid projects: Astro-Grid and LSST. Providing a solution to each of the workflow scenarios involved utilising different features of the MASC language, the simplest being the batch processing through to the most complex, the LSST runtime coordination. A concrete XML representation, used as input to the Zorro framework can be found in each of the relevant appendices. An original aim of the thesis was to provide a language that met the requirements of scientific workflow, addressed by Chapters 3, 4 and 5. Features of the MASC language were highlighted which solved each of the motivating requirements. This was following by a discussion of the features which enabled the MASC approach to solve a new class of workflow requiring flexible, runtime service composition. The following Chapter discusses concrete conclusions and the further avenues of research which could be pursed as a result of this thesis.

Chapter 8

Conclusions and Further Work

This Chapter concludes the thesis by presenting a summary of the research and highlighting the contributions to knowledge it has made, Section 8.1. Avenues for further research are discussed in Section 8.2.

8.1 Summary and Contributions to Knowledge

A problem with workflow specifications is that often the patterns of interaction between the distributed services are too complicated to predict and analyse at design-time. In certain cases, the exact patterns of message exchange and the concrete services to call cannot be predicted in advance, due to factors such as fluctuating network load or the availability of services. It is a more realistic assumption to endow software components with the ability to make decisions about the nature and scope of their interactions at runtime.

In order to facilitate flexible, runtime service composition this thesis has presented an investigation into fusing the agency and service-oriented architecture paradigms. This investigation was composed from multiple steps and made the following contributions to knowledge:

- **Deriving the requirements of scientific workflow:** By working closely with the AstroGrid project a number of concrete, realistic workflow scenarios have evolved. Scenario 1: Batch processing and scenario 2: Knowledge acquisition were presented in detail by Sections 3.2 and Section 3.3. Together with the detailed analysis of existing systems (discussed in Section 2.2), these scenarios helped derive a core set of requirements for scientific workflow; these requirements were detailed in Section 3.4. This analysis process confirmed that scientific workflow has an extra set of requirements which go beyond the functionality that traditional workflow languages and execution engines provide.

- **Counterexample scenario:** This thesis also worked closely with a second Grid project, the Large Synoptic Survey Telescope (LSST). By working with this project a detailed workflow scenario evolved which acted as a counterexample of coordination which is difficult or impossible to achieve by existing service composition techniques. This counterexample scenario was discussed in Section 4.1 and highlighted that statically defined, pre-designed/pre-planned workflows were too brittle for a scenario which required dynamic, runtime coordination, by decentralised autonomous, reactive software components. This counterexample scenario backed up the hypothesis that workflow specifications are often too complex to analyse at design-time.

- **Combined requirements:** Through a combined process of analysing existing systems and working closely with domain scenarios, Section 5.1 identified a set of desirable properties for a workflow language. This process served as the requirements analysis for the remainder of the research presented by this thesis. These combined requirements captured the essence of scientific workflow but also demanded flexible, runtime composition of services in an inherently decentralised, peer-to-peer architecture; traits which are not common of existing service composition techniques.

- **Uniting agents and services:** As discussed in detail by Chapter 2, service-oriented architectures and multiagent systems offer complementary paradigms for building distributed systems. In order to achieve the combined set of requirements for workflow this thesis viewed the service composition problem in a fundamentally different way. An agent-based architecture was proposed, allowing active, autonomous agents to consume the passive service-oriented architectures found in Internet and Grid systems.

- **Service composition through interaction protocols:** Our agent-based approach to workflow composition was founded on the concept of shared interaction protocols that allow groups of decentralised agents to communicate in open systems.

- **MultiAgent Service Composition (MASC):** Based on this concept of shared interaction protocols Chapter 5 presented an agent-based workflow language: MultiAgent Service Composition, or MASC for short. This language extended the Electronic Institutions framework and focused solely on service composition to meet the requirements analysis presented by Chapters 3 and 4. Agents acts as proxies or stubs to service invocation and can connect from the protocol code, describing the coordination model to internal reasoning models. In contrast with statically defined, centralised workflows, MASC allows decentralised agents to perform service composition at runtime, allowing them to operate in scenarios where it is not possible to define the pattern of interaction in advance. A dataflow layer allowed our agent-based coordination mechanism to be wrapped up into more complex workflows.

- **Agent-Based web services composition framework:** Chapter 6 presented the Zorro framework, an open-source Java implementation of the MASC language. This framework served as a test bed for the ideas addressed by this thesis, allowing real protocols to be executed with real services on real data.

- **Coordination-oriented programming methodology:** In addition to the MASC language and Zorro framework, Chapter 7 proposed a methodology outlining how users can build workflows using an agent-based approach to service composition. The methodology allows users with different skills and motivations to approach the system from various levels of abstraction. Users can adopt the role of *experiment engineers*, *interaction engineers* and *agent engineers*.

- **Evaluation by use-Case:** In order to demonstrate and evaluate the agent-based technique proposed by this thesis each of the motivating workflow scenarios was designed using the coordination-oriented methodology, implemented using the MASC language and executed on the Zorro framework. This process was described in detail by Chapter 7 appendices B-D contain the XML input used to execute the Zorro framework.

- **Application to live Grid project:** Workflow scenarios have been a driving factor behind this thesis. Modelling these scenarios has allowed the language and framework to evolve and provided the project with a realistic application domain. AstroGrid has served as a test bed, in order to verify and execute our ideas on a live framework, with live services and data.

8.2 Further Work

There are several avenues for further research based on the work of this thesis. Most further work involves development of the Zorro framework, which served as a prototype to facilitate the research presented by this thesis and is merely a proof-of-concept. Detailed below are the possible avenues for further research:

- **Framework development - distributed agents:** In the prototype framework, agents execute a protocol as a closely coupled system, each agent is implemented as a separate thread within a multi-threaded system. A simple extension would allow a number of distributed agents to execute a scene definition, instead of running each agent as a separate thread on the same server. A user could then chose whether to execute agents locally, as distributed processes, or a combination of both. This would result in two fundamental differences, the first is that agents are not dynamically created within a scene, they are located and initialised and executed across a network. Secondly, message passing takes places through a distributed protocol (such as SOAP), instead of exchanging messages between a multi-threaded system.

- **Framework development - visual protocol builder tool:** Designing protocols which define how agent roles interact with one another (the task of an interaction engineer) is a complex, error prone task. A front-end, visualisation tool to aid an interaction engineer could prove a more efficient method of protocol design. This front-end would allow a user to create a protocol by visualising the design process, dragging, dropping and editing components, this could then be translated to the formal specification for execution.

- **Framework development - user interaction:** Through close analysis of scenarios and existing systems, user interaction has emerged as a core requirement for scientific workflow. The MASC language has several features which facilitate this requirement, discussed in more detail by Sections 5.2.3.3 and 5.2.6. The prototype framework has implemented several of these features, however to make the framework useable for real domain scientists, additional tool support is required.

- **Framework development - tool integration:** This thesis has not intended to reinvent the wheel, although scientific workflow is a relatively new field, maturing scientific workflow systems exist. By wrapping our agent-based approach to service composition in a dataflow layer, it is possible to integrate models of agency to existing, mature scientific workflow systems, such as Taverna [43]. Although this thesis has discussed the possibility of tool integration, the framework needs several simple additions to facilitate this functionality.

- **Scenario development:** The process of working with the Virtual Observatory community has been a two way process. Scenario modelling has influenced the requirements for this research and in return agent-based techniques have proposed a solution to open coordination problems within this domain. The Virtual Observatory domain was chosen because of the interesting coordination challenges faced by scientists, however other equally interesting domains exist where agent-based workflow techniques would be applicable.
- **Integration of complex agent reasoning:** This thesis was primarily focused on developing techniques for flexible service composition. The evaluation demonstrated how these techniques could be deployed to build workflows. Simple reasoning was integrated into the knowledge acquisition and runtime coordination scenarios, however it would be an interesting exercise to include more complex models of agent reasoning into the decision procedures, such as the Belief Desires and Intentions (BDI) model.
- **Startup issues:** There are a number of unsolved issues regarding how to locate and disseminate a protocol to a group of distributed agents. Currently all agents are executed locally within a scene process, so the problems of agent location, protocol dissemination and agent initialisation are avoided. Startup algorithms need to be developed to solve these issues, some of which are being addressed by the currently running OpenKnowledge project [57].

Appendix A

MultiAgent Service Composition (MASC) XML Schema Definition

```
<?xml version="1.0" encoding="UTF-8" ?>

<xsd:schema
        targetNamespace="urn:zorro"
        elementFormDefault="qualified"
        xmlns="urn:zorro"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            ///////////////////////////////////////
            // //
            // Zorro Framework //
            // File: basic.xsd //
            // Adam Barker - Copyright (C) 2006 //
            ///////////////////////////////////////
        </xsd:documentation>
    </xsd:annotation>

    <!-- Root Element -->
    <xsd:element name="protocol" type="protocoldefinition"/>

    <!-- Protocol Type, Base Type -->
    <xsd:complexType name="protocoldefinition">

        <!-- Overall Input and Output to the Experiment -->
        <xsd:sequence>

            <!-- Scene Type -->
            <xsd:element name="sceneset" type="Scene" minOccurs="0" maxOccurs="1"/>

            <xsd:element name="mapping" minOccurs="0" maxOccurs="1">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="node" type="node" minOccurs="1" maxOccurs="unbounded"/>
                        <xsd:element name="link" type="link" minOccurs="0" maxOccurs="unbounded"/>
                    </xsd:sequence>
                    <xsd:attribute name="name" type="xsd:Name"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:Name"/>
    </xsd:complexType>
```

```
<!-- A Link -->
<xsd:complexType name="link">
    <xsd:sequence>
        <xsd:element name="source" minOccurs="1" maxOccurs="1">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="mappinglinksource" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="sink" minOccurs="1" maxOccurs="1">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="mappinglinksink" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<!-- Scene Type -->
<xsd:complexType name="SceneType">
    <xsd:sequence>

        <!-- Scene In-Ports -->
        <xsd:element name="input" minOccurs="0" maxOccurs="1">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="port" type="portdefinition" minOccurs="1" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <!-- Scene Out-Ports -->
        <xsd:element name="output" minOccurs="0" maxOccurs="1">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="port" type="portdefinition" minOccurs="1" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <!-- Process Definition -->
        <xsd:element name="agent" type="process" maxOccurs="unbounded"/>

    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:Name"/>
</xsd:complexType>

<xsd:complexType name="process">
    <xsd:sequence>
        <!-- Process Input A Process can be parameterised by the user -->
        <xsd:element name="processinput" type="IO" minOccurs="0" maxOccurs="1"/>

        <!-- Method Definition -->
        <xsd:element name="method" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                    <!-- Can be removed -->
                    <xsd:element name="in" minOccurs="0" maxOccurs="1">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:group ref="type" minOccurs="0" maxOccurs="unbounded"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                    <!-- Can be removed -->
                    <xsd:element name="out" minOccurs="0" maxOccurs="1">
                        <xsd:complexType>
                            <xsd:sequence>
```

```
                                <xsd:group ref="type" minOccurs="0" maxOccurs="unbounded"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                    <xsd:element name="body" type="SequenceType"/>
                </xsd:sequence>
                <xsd:attribute name="name" type="xsd:Name"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>

    <!-- Parameters for the Agent Type -->
    <xsd:attribute name="role" type="xsd:string"/>
    <xsd:attribute name="implementation" type="xsd:string" use="optional"/>
    <xsd:attribute name="min" type="xsd:nonNegativeInteger"
                   use="optional"/>
    <xsd:attribute name="max" type="allNNI" use="optional"/>
</xsd:complexType>

<!-- Simple Type which is either an non-negative integer or unbounded -->
<xsd:simpleType name="allNNI">
    <xsd:union memberTypes="xsd:nonNegativeInteger">
        <xsd:simpleType>
            <xsd:restriction base="xsd:NMTOKEN">
                <xsd:enumeration value="unbounded"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:union>
</xsd:simpleType>

<!-- Sequences -->
<xsd:complexType name="SequenceType">
    <xsd:sequence>
        <xsd:group ref="Operation" minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<!-- Operations -->
<xsd:group name="Operation">
    <xsd:choice>

        <!-- Action -->
        <xsd:group ref="Action"/>

        <!-- Choice  -->
        <xsd:element name="choice">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="op" type="SequenceType"
                                 minOccurs="2" maxOccurs="2"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <!-- Parallel Execution -->
        <xsd:element name="par">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="par" type="SequenceType"
                                 minOccurs="2" maxOccurs="2"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <!-- Loop -->
        <xsd:element name="while">
            <xsd:complexType>
                <xsd:sequence>
                    <!-- Constraint placed on the Loop -->
                    <xsd:element name="constraint" type="single_constraint" minOccurs="0" maxOccurs="1"/>
                    <xsd:element name="body" type="SequenceType"/>
                    <xsd:element name="timeout" type="SequenceType"/>
                </xsd:sequence>
                <xsd:attribute name="tmax" type="xsd:nonNegativeInteger" use="optional"/>
            </xsd:complexType>
        </xsd:element>

        <!-- Recursive Call -->
        <xsd:element name="call">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="in" minOccurs="0" maxOccurs="1">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:group ref="type" minOccurs="0" maxOccurs="unbounded"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                    <xsd:element name="out" minOccurs="0" maxOccurs="1">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:group ref="type" minOccurs="0" maxOccurs="unbounded"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
                <xsd:attribute name="name" type="xsd:Name"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>

<!-- Actions -->
<xsd:group name="Action">
    <xsd:choice>
        <!-- Decision Procedure -->
        <xsd:element name="proc" type="proc"/>

        <!-- Web Service Invocation -->
        <xsd:element name="service" type="wproc"/>

        <!-- Constraint -->
        <xsd:element name="cproc" type="single_constraint"/>

        <!-- Send Message -->
        <xsd:element name="send">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="mesg" type="MesgType"/>
                    <xsd:element name="clause1">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:group ref="clause1" minOccurs="1" maxOccurs="1"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>

                    <xsd:element name="clause2">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:group ref="clause2" minOccurs="1" maxOccurs="1"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                    <!-- *** Sending Constraint -->
                    <xsd:element name="constraint" type="single_constraint" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <!-- Port Write -->
        <xsd:element name="portwrite">
            <xsd:complexType>
                <xsd:sequence>
```

```
                    <xsd:group ref="type" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="name" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>

        <!-- Receive Message -->
        <xsd:element name="recv">
            <xsd:complexType>
                <xsd:sequence>
                    <!-- *** Receiving Constraint -->
                    <xsd:element name="constraint" type="single_constraint" minOccurs="0" maxOccurs="1"/>
                    <xsd:element name="mesg" type="MesgType"/>

                    <xsd:element name="clause1">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:group ref="clause1" minOccurs="1" maxOccurs="1"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>

                    <xsd:element name="clause2">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:group ref="clause2" minOccurs="1" maxOccurs="1"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <!-- UnPack Operation -->
        <xsd:element name="unpack">
            <xsd:complexType>
                <xsd:group ref="type" minOccurs="1" maxOccurs="1"/>
            </xsd:complexType>
        </xsd:element>

        <!-- Pack Operation -->
        <xsd:element name="pack">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="Scene" type="SceneType" minOccurs="1" maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="packref" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>

        <!-- Null Action -->
        <xsd:element name="null"/>
    </xsd:choice>
</xsd:group>

<!-- Procedure -->
<xsd:complexType name="proc">
    <xsd:sequence>
        <xsd:element name="in" minOccurs="0" maxOccurs="1">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="type" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="out" minOccurs="0" maxOccurs="1">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="type" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:Name"/>
</xsd:complexType>

<!-- Service Invocation -->
<xsd:complexType name="wproc">
    <xsd:sequence>
        <xsd:element name="def" type="webservice" minOccurs="1" maxOccurs="1"/>

        <xsd:element name="in" minOccurs="0" maxOccurs="1">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="type" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="out" minOccurs="0" maxOccurs="1">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="type" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

    </xsd:sequence>
</xsd:complexType>

<!-- Constraints -->
<xsd:complexType name="single_constraint">
    <xsd:choice>
        <xsd:element name="proc" type="proc"/>
        <xsd:element name="wproc" type="wproc"/>
    </xsd:choice>
    <xsd:attribute name="name" type="xsd:Name"/>
</xsd:complexType>

<!-- Messages -->
<xsd:complexType name="MesgType">
    <xsd:sequence>
        <xsd:group ref="type" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="performative" type="xsd:string"/>
</xsd:complexType>

<!-- Clause one -->
<xsd:group name="clause1">
    <xsd:choice>
        <!-- Agent ID -->
        <xsd:element name="agent">
            <xsd:complexType>
                <xsd:attribute name="name" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>

        <!-- Variable Representing an Agent -->
        <xsd:group ref="type" minOccurs="0" maxOccurs="1"/>

        <!-- Wild Card -->
        <xsd:element name="wild"/>
    </xsd:choice>
</xsd:group>

<!-- Clause two -->
<xsd:group name="clause2">
    <xsd:choice>
        <xsd:element name="role">
            <xsd:complexType>
                <xsd:attribute name="name" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="wild"/>
    </xsd:choice>
</xsd:group>

<!-- Type System -->
<xsd:group name="type">
    <xsd:choice>

        <!-- Reading from a Port -->
        <xsd:element name="portread">
            <xsd:complexType>
                <xsd:attribute name="name" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>

        <!-- Variable Type -->
        <xsd:element name="var">
            <xsd:complexType>
                <xsd:attribute name="name" type="xsd:string"/>
                <xsd:attribute name="type" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>

        <!-- Constant Type -->
        <xsd:element name="const">
            <xsd:complexType>
                <xsd:attribute name="value" type="xsd:string"/>
                <xsd:attribute name="type" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>

<!-- Agent Type -->
<xsd:complexType name="Agent">
    <xsd:attribute name="name" type="xsd:Name"/>
    <xsd:attribute name="implementation" type="xsd:string"/>
</xsd:complexType>

<!-- Port Definition -->
<xsd:complexType name="portdefinition">
    <xsd:sequence>
        <xsd:element name="constraint" type="single_constraint" minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="core" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- Input/Output Type -->
<xsd:complexType name="IO">
    <xsd:sequence>
        <xsd:group ref="type" minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<!-- Source of the Mapping Link -->
<xsd:group name="mappinglinksource">
    <xsd:choice>
        <xsd:element name="outport">
            <xsd:complexType>
                <xsd:attribute name="scene" type="xsd:string"/>
                <xsd:attribute name="port" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="user"/>

        <xsd:element name="file">
            <xsd:complexType>
                <xsd:attribute name="location" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="application"/>

        <xsd:element name="webservice" type="webservice" minOccurs="1" maxOccurs="1"/>
    </xsd:choice>
</xsd:group>

<!-- Sink of the Mapping Link -->
<xsd:group name="mappinglinksink">
    <xsd:choice>
        <xsd:element name="inport">
            <xsd:complexType>
                <xsd:attribute name="scene" type="xsd:string"/>
                <xsd:attribute name="port" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="user">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="mappinglinksink" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="file">
            <xsd:complexType>
                <xsd:attribute name="location" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="application"/>

        <xsd:element name="webservice" type="webservice" minOccurs="1" maxOccurs="1"/>
    </xsd:choice>
</xsd:group>

<!-- Web Service Definition -->
<xsd:complexType name="webservice">
    <xsd:attribute name="wsdl" type="xsd:anyURI"/>
    <xsd:attribute name="service" type="xsd:Name"/>
    <xsd:attribute name="port" type="xsd:Name"/>
    <xsd:attribute name="namespace" type="xsd:anyURI"/>
    <xsd:attribute name="opname" type="xsd:Name"/>
</xsd:complexType>

<xsd:complexType name="node">
    <xsd:sequence>
        <xsd:element name="role" type="roleInformation" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="location" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="roleInformation">
    <xsd:sequence>
        <xsd:element name="agent" type="agentInformation" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>

<!-- Paramterisable Agent Type -->
<xsd:complexType name="agentInformation">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="implementation" type="xsd:anyURI" use="required"/>
    <xsd:attribute name="num" type="xsd:int" use="required"/>
    <xsd:attribute name="recvwait" type="xsd:int" use="required"/>
    <xsd:attribute name="portwait" type="xsd:int" use="required"/>
</xsd:complexType>
```

```
    <xsd:complexType name="Scene">
        <xsd:sequence>
            <xsd:element name="Scene" type="SceneType" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Appendix B

XML Implementation of Scenario 1: Batch Processing

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<protocol xmlns="urn:zorro">
    <sceneset>
        <Scene name="redshift">
            <!-- RSM Agent -->
            <agent implementation="http://localhost:8080/MyRsmService/rsm?WSDL" max="1" min="1" role="rsm">
                <!-- MAIN Method -->
                <method name="main">
                    <body>
                        <while>
                            <body>
                                <!-- Receive RA and DEC from USER agent -->
                                <recv>
                                    <mesg performative="request">
                                        <var name="ra"/>
                                        <var name="dec"/>
                                    </mesg>
                                    <clause1>
                                        <var name="user_config"/>
                                    </clause1>
                                    <clause2>
                                        <role name="user"/>
                                    </clause2>
                                </recv>
                            </body>
                            <timeout>
                                <null/>
                            </timeout>
                        </while>

                        <!-- Invoke Retrieve Method -->
                        <call name="retrieve">
                            <in>
                                <var name="ra"/>
                                <var name="dec"/>
                                <var name="user_config"/>
                            </in>
                        </call>

                        <!-- Restart Agent -->
                        <call name="main"/>
                    </body>
                </method>
```

```
<!-- RETRIEVE Method -->
<method name="retrieve">
    <in>
        <var name="ra"/>
        <var name="dec"/>
        <var name="user_config"/>
    </in>
    <out/>
    <body>
        <!-- Retrieve Images -->
        <service>
            <def
                    namespace="urn:Foo"
                    opname="getImages"
                    port="RsmIFPort"
                    service="MyRsmService"
                    wsdl="http://localhost:8080/MyRsmService/rsm?WSDL"/>
            <in>
                <var name="ra"/>
                <var name="dec"/>
            </in>
            <out>
                <var name="images" type="xsd:string"/>
            </out>
        </service>

        <!-- Sextractor -->
        <service>
            <def
                    namespace="urn:Foo"
                    opname="sextractor"
                    port="RsmIFPort"
                    service="MyRsmService"
                    wsdl="http://localhost:8080/MyRsmService/rsm?WSDL"/>
            <in>
                <var name="images"/>
            </in>
            <out>
                <var name="sextractor" type="xsd:string"/>
            </out>
        </service>

        <!-- X-Matching -->
        <service>
            <def
                    namespace="urn:Foo"
                    opname="xmatcher"
                    port="RsmIFPort"
                    service="MyRsmService"
                    wsdl="http://localhost:8080/MyRsmService/rsm?WSDL"/>
            <in>
                <var name="sextractor"/>
            </in>
            <out>
                <var name="xmatched" type="xsd:string"/>
            </out>
        </service>

        <!-- Invoke RedShift Method -->
        <call name="redshift">
            <in>
                <var name="xmatched"/>
                <var name="user_config"/>
            </in>
        </call>
    </body>
</method>

<!-- REDSHIFT Method -->
<method name="redshift">
    <in>
```

```
            <var name="xmatched"/>
            <var name="user_config"/>
        </in>
        <body>
            <!-- Sextractor -->
            <service>
                <def
                        namespace="urn:Foo"
                        opname="hyperz"
                        port="RsmIFPort"
                        service="MyRsmService"
                        wsdl="http://localhost:8080/MyRsmService/rsm?WSDL"/>
                <in>
                    <var name="xmatched"/>
                </in>
                <out>
                    <var name="hyperz" type="xsd:string"/>
                </out>
            </service>

            <!-- Send results back to the USER Agent -->
            <send>
                <mesg performative="response">
                    <var name="hyperz"/>
                </mesg>
                <clause1>
                    <var name="user_config"/>
                </clause1>
                <clause2>
                    <role name="user"/>
                </clause2>
            </send>
        </body>
    </method>
</agent>

<!-- USER Agent -->
<agent implementation="http://localhost:8080/MyRsmService/user?WSDL" max="1" min="1" role="user">
    <!-- MAIN Method -->
    <method name="main">
        <body>
            <!-- Get RA -->
            <proc name="sayRaDec">
                <out>
                    <var name="ra"/>
                </out>
            </proc>

            <!-- Get DEC -->
            <proc name="sayRaDec">
                <out>
                    <var name="dec"/>
                </out>
            </proc>

            <!-- Send RA and DEC to RSM Agent -->
            <send>
                <mesg performative="request">
                    <var name="ra"/>
                    <var name="dec"/>
                </mesg>
                <clause1>
                    <wild/>
                </clause1>
                <clause2>
                    <role name="rsm"/>
                </clause2>
            </send>

            <!-- Call WAIT Method -->
            <call name="wait"/>
        </body>
```

```
                </method>

                <!-- WAIT Method -->
                <method name="wait">
                    <body>
                        <while>
                            <body>
                                <!-- Receive results from RSM Agent -->
                                <recv>
                                    <mesg performative="response">
                                        <var name="hyperz"/>
                                    </mesg>
                                    <clause1>
                                        <wild/>
                                    </clause1>
                                    <clause2>
                                        <role name="rsm"/>
                                    </clause2>
                                </recv>
                            </body>
                            <timeout>
                                <null/>
                            </timeout>
                        </while>
                    </body>
                </method>
            </agent>
        </Scene>
    </sceneset>
</protocol>
```

Appendix C

XML Implementation of Scenario 2: Knowledge Acquisition

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<protocol xmlns="urn:zorro">
    <sceneset>
        <Scene name="bcg">
            <!-- SCIENTIST agent -->
            <agent implementation="http://localhost:8080/MyScientistService/scientist?WSDL" max="1" min="1"
                   role="scientist">
                <!--MAIN method -->
                <method name="main">
                    <body>
                        <while>
                            <body>
                                <!-- Receive initial request from USER -->
                                <recv>
                                    <mesg performative="begin"/>
                                    <clause1>
                                        <var name="user_config"/>
                                    </clause1>
                                    <clause2>
                                        <role name="user"/>
                                    </clause2>
                                </recv>

                                <!-- Invoke DECISION PROCEDURE to determine object class -->
                                <proc name="NextLookUp">
                                    <out>
                                        <var name="query"/>
                                    </out>
                                </proc>

                                <!-- Registry Lookup on Galaxies -->
                                <service>
                                    <def
                                            namespace="urn:Foo"
                                            opname="Registry"
                                            port="ScientistIFPort"
                                            service="MyScientistService"
                                            wsdl="http://localhost:8080/MyScientistService/scientist?WSDL"/>
                                    <in>
                                        <var name="query"/>
                                    </in>
                                    <out>
                                        <var name="galaxies" type="xsd:string"/>
                                    </out>
                                </service>

                                <!-- Send extraction request to EXTRACTION agent -->
                                <send>
                                    <mesg performative="extract">
                                        <var name="galaxies"/>
                                    </mesg>
                                    <clause1>
                                        <wild/>
                                    </clause1>
                                    <clause2>
                                        <role name="extraction"/>
                                    </clause2>
                                </send>

                                <!-- Invoke WAIT Method -->
                                <call name="wait">
                                    <in>
                                        <var name="user_config"/>
                                    </in>
                                </call>
                            </body>
                            <timeout>
                                <null/>
                            </timeout>
                        </while>
                    </body>
                </method>

                <!-- WAIT method -->
                <method name="wait">
                    <in>
                        <var name="user_config"/>
                    </in>
                    <body>
                        <while tmax="75">
                            <body>
                                <!-- Receive query results from EXTRACTION agent -->
                                <recv>
                                    <mesg performative="finalresult">
                                        <var name="galaxy_data"/>
                                    </mesg>
                                    <clause1>
                                        <wild/>
                                    </clause1>
                                    <clause2>
                                        <role name="extraction"/>
                                    </clause2>
                                </recv>
                            </body>
                            <timeout>
                                <null/>
                            </timeout>
                        </while>

                        <!-- Invoke DECISION PROCEDURE to determine object class -->
                        <proc name="NextLookUp">
                            <out>
                                <var name="query"/>
                            </out>
                        </proc>

                        <!-- Registry Lookup on Radio etc. -->
                        <service>
                            <def
                                    namespace="urn:Foo"
                                    opname="Registry"
                                    port="ScientistIFPort"
                                    service="MyScientistService"
                                    wsdl="http://localhost:8080/MyScientistService/scientist?WSDL"/>
                            <in>
```

```
                <var name="query"/>
            </in>
            <out>
                <var name="extra" type="xsd:string"/>
            </out>
        </service>

        <!-- Send extraction request to EXTRACTION agent -->
        <send>
            <mesg performative="extract">
                <var name="extra"/>
            </mesg>
            <clause1>
                <wild/>
            </clause1>
            <clause2>
                <role name="extraction"/>
            </clause2>
        </send>

        <!-- Receive Results -->
        <while tmax="30">
            <body>
                <!-- Receive Final Result -->
                <recv>
                    <mesg performative="finalresult">
                        <var name="extra_data"/>
                    </mesg>
                    <clause1>
                        <wild/>
                    </clause1>
                    <clause2>
                        <role name="extraction"/>
                    </clause2>
                </recv>
                <!-- Invoke Decision Procedure -->
                <call name="bcg">
                    <in>
                        <var name="galaxy_data"/>
                        <var name="extra_data"/>
                        <var name="user_config"/>
                    </in>
                </call>
            </body>
            <timeout>
                <null/>
            </timeout>
        </while>
    </body>
</method>

<!-- BCG Method -->
<method name="bcg">
    <in>
        <var name="galaxy_data"/>
        <var name="extra_data"/>
        <var name="user_config"/>
    </in>
    <body>
        <!-- Calculate Working Data Set -->
        <service>
            <def
                    namespace="urn:Foo"
                    opname="Stats1"
                    port="ScientistIFPort"
                    service="MyScientistService"
                    wsdl="http://localhost:8080/MyScientistService/scientist?WSDL"/>
            <in>
                <var name="galaxy_data"/>
                <var name="extra_data"/>
            </in>
            <out>
```

```
                <var name="working_set" type="xsd:string"/>
            </out>
        </service>

        <!-- Top Attributes -->
        <service>
            <def
                    namespace="urn:Foo"
                    opname="Stats2"
                    port="ScientistIFPort"
                    service="MyScientistService"
                    wsdl="http://localhost:8080/MyScientistService/scientist?WSDL"/>
            <in>
                <var name="working_set"/>
            </in>
            <out>
                <var name="top_attributes" type="xsd:string"/>
            </out>
        </service>

        <!-- Send To USER -->
        <send>
            <mesg performative="display">
                <var name="top_attributes"/>
                <var name="galaxy_data"/>
                <var name="extra_data"/>
            </mesg>
            <clause1>
                <var name="user_config"/>
            </clause1>
            <clause2>
                <role name="user"/>
            </clause2>
        </send>

        <!-- Invoke USER INTERACTION -->
        <call name="userinteraction">
            <in>
                <var name="user_config"/>
            </in>
        </call>
    </body>
</method>

<!-- USERINTERACTION Method -->
<method name="userinteraction">
    <in>
        <var name="user_config"/>
    </in>
    <body>
        <while>
            <body>
                <choice>
                    <op>
                        <!-- Receive further interaction request -->
                        <recv>
                            <mesg performative="visualisation">
                                <var name="parameter_list"/>
                            </mesg>
                            <clause1>
                                <wild/>
                            </clause1>
                            <clause2>
                                <role name="user"/>
                            </clause2>
                        </recv>

                        <!-- Invoke STATISTICS Web Service -->
                        <service>
                            <def
                                    namespace="urn:Foo"
                                    opname="Stats1"
```

```
                                        port="ScientistIFPort"
                                        service="MyScientistService"
                                        wsdl="http://localhost:8080/MyScientistService/scientist?WSDL"/>
                                <in>
                                    <var name="parameter_list"/>
                                </in>
                                <out>
                                    <var name="result" type="xsd:string"/>
                                </out>
                            </service>

                            <!-- Send display results to USER Agent -->
                            <send>
                                <mesg performative="display">
                                    <var name="result"/>
                                </mesg>
                                <clause1>
                                    <var name="user_config"/>
                                </clause1>
                                <clause2>
                                    <role name="user"/>
                                </clause2>
                            </send>
                        </op>
                        <!-- OR Choice -->
                        <op>
                            <!-- Receive the Termination message from USER agent -->
                            <recv>
                                <mesg performative="finished">
                                </mesg>
                                <clause1>
                                    <wild/>
                                </clause1>
                                <clause2>
                                    <role name="user"/>
                                </clause2>
                            </recv>
                            <!-- Start the scientist again -->
                            <call name="main"/>
                        </op>
                    </choice>
                </body>
                <timeout>
                    <null/>
                </timeout>
            </while>
        </body>
    </method>
</agent>

<!-- EXTRACTION agent -->
<agent implementation="http://localhost:8080/MyExtractionService/extraction?WSDL" max="1" min="10"
       role="extraction">
    <!-- MAIN method-->
    <method name="main">
        <body>
            <while>
                <body>
                    <!-- Receive extraction request from SCIENTIST agent -->
                    <recv>
                        <mesg performative="extract">
                            <var name="qlist"/>
                        </mesg>
                        <clause1>
                            <var name="scientist"/>
                        </clause1>
                        <clause2>
                            <role name="scientist"/>
                        </clause2>
                    </recv>

                    <proc name="Initialise">
                    <out>
                        <var name="result"/>
                    </out>
                </proc>
            </body>
            <timeout>
                <null/>
            </timeout>
        </while>

        <!-- Invoke ELOOP Method -->
        <call name="eloop">
            <in>
                <var name="qlist"/>
                <var name="scientist"/>
            </in>
        </call>

        <!-- Invoke MAIN Method -->
        <call name="main"/>
    </body>
</method>

<!-- ELOOP Method -->
<method name="eloop">
    <in>
        <var name="qlist"/>
        <var name="scientist"/>
    </in>
    <body>
        <choice>
            <op>
                <!-- Extract head of List -->
                <proc name="Head">
                    <in>
                        <var name="qlist"/>
                    </in>
                    <out>
                        <var name="head"/>
                    </out>
                </proc>

                <!-- Extract tail of List -->
                <proc name="Tail">
                    <out>
                        <var name="tail"/>
                    </out>
                </proc>

                <!-- Construct Query type -->
                <proc name="ConstructQuery">
                    <in>
                        <var name="head"/>
                    </in>
                    <out>
                        <var name="q"/>
                    </out>
                </proc>

                <!-- Send Query -->
                <send>
                    <mesg performative="query">
                        <var name="q"/>
                    </mesg>
                    <clause1>
                        <wild/>
                    </clause1>
                    <clause2>
                        <role name="endpoint"/>
                    </clause2>
                </send>
                <!-- Invoke ELOOP Method  -->
                <call name="eloop">
                    <in>
                        <var name="tail"/>
                        <var name="scientist"/>
                    </in>
                </call>
            </op>

            <!-- OR Choice -->
            <op>
                <!-- Invoke EWAIT Method -->
                <call name="ewait">
                    <in>
                        <var name="scientist"/>
                    </in>
                </call>
            </op>
        </choice>
    </body>
</method>

<!-- EWAIT Method -->
<method name="ewait">
    <in>
        <var name="scientist"/>
    </in>
    <body>
        <while>
            <body>
                <choice>
                    <op>
                        <!-- Receive results from ENDPOINT agent -->
                        <recv>
                            <mesg performative="result">
                                <var name="res"/>
                            </mesg>
                            <clause1>
                                <var name="name"/>
                            </clause1>
                            <clause2>
                                <role name="endpoint"/>
                            </clause2>
                        </recv>

                        <!-- Store the Result -->
                        <proc name="Store">
                            <in>
                                <var name="name"/>
                                <var name="res"/>
                            </in>
                        </proc>

                        <!-- Recursive CALL on EWAIT Method -->
                        <call name="ewait">
                            <in>
                                <var name="scientist"/>
                            </in>
                        </call>
                    </op>
                    <!-- OR Choice -->
                    <op>
                        <!-- Receive NORESULT message from ENDPOINT agent -->
                        <recv>
                            <mesg performative="noresult">
                            </mesg>
                            <clause1>
                                <var name="name"/>
                            </clause1>
                            <clause2>
                                <role name="endpoint"/>
                            </clause2>
                            </recv>

                            <!-- Recursive CALL on EWAIT Method -->
                            <call name="ewait">
                                <in>
                                    <var name="scientist"/>
                                </in>
                            </call>
                        </op>
                    </choice>
                </body>
                <timeout>
                    <!-- Invoke the END Method -->
                    <call name="end">
                        <in>
                            <var name="scientist"/>
                        </in>
                    </call>
                </timeout>
            </while>
        </body>
    </method>

    <!-- END Method -->
    <method name="end">
        <in>
            <var name="scientist"/>
        </in>
        <body>
            <!-- Retrieve the Result -->
            <proc name="Retrieve">
                <out>
                    <var name="data"/>
                </out>
            </proc>

            <!-- Store the results in MYSPACE -->
            <service>
                <def
                        namespace="urn:Foo"
                        opname="MySpace"
                        port="ExtractionIFPort"
                        service="MyExtractionService"
                        wsdl="http://localhost:8080/MyExtractionService/extraction?WSDL"/>
                <in>
                    <var name="data"/>
                </in>
                <out>
                    <var name="resulturl" type="xsd:string"/>
                </out>
            </service>

            <!-- Send the Results back to the original SCIENTIST -->
            <send>
                <mesg performative="finalresult">
                    <var name="resulturl"/>
                </mesg>
                <clause1>
                    <var name="scientist"/>
                </clause1>
                <clause2>
                    <role name="scientist"/>
                </clause2>
            </send>
        </body>
    </method>
</agent>

<!-- ENDPOINT agent -->
<agent implementation="http://localhost:8080/MyEndService/end?WSDL" max="1" min="10" role="endpoint">
    <!-- MAIN Method -->
    <method name="main">
```

```
        <body>
            <while>
                <body>
                    <!-- Receive message from EXTRACTION agent -->
                    <recv>
                        <mesg performative="query">
                            <var name="queryToTry"/>
                        </mesg>
                        <clause1>
                            <var name="requester"/>
                        </clause1>
                        <clause2>
                            <role name="extraction"/>
                        </clause2>
                    </recv>

                    <!-- Invoke QUERY Method -->
                    <call name="query">
                        <in>
                            <var name="requester"/>
                            <var name="queryToTry"/>
                        </in>
                    </call>
                </body>
                <timeout>
                    <null/>
                </timeout>
            </while>
        </body>
    </method>

    <!-- QUERY Method -->
    <method name="query">
        <in>
            <var name="requester"/>
            <var name="queryToTry"/>
        </in>
        <body>
            <!-- Query Success -->
            <proc name="Success">
                <in>
                    <var name="queryToTry"/>
                </in>
                <out>
                    <var name="success"/>
                </out>
            </proc>

            <!-- Send RESULT -->
            <send>
                <mesg performative="result">
                    <var name="success"/>
                </mesg>
                <clause1>
                    <var name="requester"/>
                </clause1>
                <clause2>
                    <role name="extraction"/>
                </clause2>
            </send>

            <!-- Invoke MAIN Method -->
            <call name="main"/>
        </body>
    </method>
</agent>

<!-- USER Agent -->
<agent implementation="http://localhost:8080/MyExtractionService/extraction?WSDL" max="1" min="1"
       role="user">
    <method name="main">
        <body>
```

```
                        <send>
                            <mesg performative="begin">
                            </mesg>
                            <clause1>
                                <wild/>
                            </clause1>
                            <clause2>
                                <role name="scientist"/>
                            </clause2>
                        </send>
                    </body>
                </method>
            </agent>
        </Scene>
    </sceneset>
</protocol>
```

Appendix D

XML Implementation of Scenario 3: Runtime Coordination

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<protocol xmlns="urn:zorro">
    <sceneset>
        <Scene name="lsst">
            <!-- Port Definitions -->
            <input>
                <port name="lsst_in1" type="xsd:string" core="true"/>
            </input>

            <!-- CLASSIFICATION Agent -->
            <agent implementation="http://localhost:8080/MyReactiveService/reactive?WSDL" max="1" min="1"
                   role="classification">
                <!-- MAIN Method -->
                <method name="main">
                    <body>
                        <while>
                            <body>
                                <!-- Receive RA and DEC from USER agent -->
                                <recv>
                                    <mesg performative="request">
                                        <var name="unknown"/>
                                    </mesg>
                                    <clause1>
                                        <wild/>
                                    </clause1>
                                    <clause2>
                                        <role name="user"/>
                                    </clause2>
                                </recv>
                            </body>
                            <timeout>
                                <null/>
                            </timeout>
                        </while>

                        <!-- Call LOCALANALYSIS Method -->
                        <call name="localanalysis">
                            <in>
                                <var name="unknown"/>
                            </in>
                        </call>
                    </body>
                </method>
```

```
<!-- RETRIEVE Method -->
<method name="localanalysis">
    <in>
        <var name="unknown"/>
    </in>
    <body>
        <choice>
            <op>
                <!-- STATTEST Decision Procedure -->
                <proc name="StatTest">
                    <in>
                        <portread name="lsst_in1"/>
                    </in>
                    <out>
                        <var name="result"/>
                    </out>
                </proc>

                <!-- UPDATE KNOWLEDGE Decision Procedure -->
                <proc name="UpdateKnowledge">
                    <in>
                        <var name="result"/>
                    </in>
                </proc>

                <!-- QUERYKNOWLEDGE Decision Procedure -->
                <proc name="QueryKnowledge">
                    <in>
                        <var name="unknown"/>
                    </in>
                    <out>
                        <var name="success"/>
                    </out>
                </proc>

                <!-- Recursive Call -->
                <call name="localanalysis"/>
            </op>

            <!-- OR Choice -->
            <op>
                <call name="contractnetsend">
                    <in>
                        <var name="unknown"/>
                    </in>
                </call>
            </op>
        </choice>
    </body>
</method>

<!-- CONTRACTNETSEND Method -->
<method name="contractnetsend">
    <in>
        <var name="unknown"/>
    </in>
    <body>
        <!-- Registry Call -->
        <service>
            <def
                    namespace="urn:Foo"
                    opname="Registry"
                    port="ReactiveIFPort"
                    service="MyReactiveService"
                    wsdl="http://localhost:8080/MyReactiveService/reactive?WSDL"/>
            <in>
                <var name="unknown"/>
            </in>
            <out>
                <var name="potential_agents"/>
            </out>
        </service>
```

```
<!-- Generate Proposal -->
<proc name="GenerateProposal">
    <in>
        <var name="unknown"/>
    </in>
    <out>
        <var name="proposal"/>
    </out>
</proc>

<!-- Send Contractnet request to CONTRACTNET Agent -->
<send>
    <mesg performative="request">
        <var name="potential_agents"/>
        <var name="proposal"/>
    </mesg>
    <clause1>
        <wild/>
    </clause1>
    <clause2>
        <role name="contractnet"/>
    </clause2>
</send>

<while tmax="30">
    <body>
        <!-- Response from CONTRACTNET Agent -->
        <recv>
            <mesg performative="response">
                <var name="open_proposals"/>
                <var name="c_id"/>
            </mesg>
            <clause1>
                <var name="cn"/>
            </clause1>
            <clause2>
                <role name="contractnet"/>
            </clause2>
        </recv>

        <!-- Agents to Reject -->
        <proc name="EvaluateReject">
            <in>
                <var name="open_proposals"/>
            </in>
            <out>
                <var name="reject"/>
            </out>
        </proc>

        <!-- Agent to Accept -->
        <proc name="EvaluateAccept">
            <in>
                <var name="open_proposals"/>
            </in>
            <out>
                <var name="accept"/>
            </out>
        </proc>

        <!-- Initialise variables -->
        <proc name="Initialise">
            <out>
                <var name="success"/>
            </out>
        </proc>

        <call name="contractreject">
            <in>
                <var name="reject"/>
                <var name="c_id"/>
```

```
                        <var name="accept"/>
                        <var name="unknown"/>
                    </in>
                </call>
            </body>
            <timeout>
                <null/>
            </timeout>
        </while>
    </body>
</method>

<!-- CONTRACTACCEPT Method -->
<method name="contractaccept">
    <in>
        <var name="accept"/>
        <var name="unknown"/>
        <var name="c_id"/>
    </in>
    <body>

        <!-- Issue Accept message to Observatory Agent -->
        <send>
            <mesg performative="accept-proposal">
                <var name="c_id"/>
            </mesg>
            <clause1>
                <var name="accept"/>
            </clause1>
            <clause2>
                <wild/>
            </clause2>
        </send>

        <while tmax="50">
            <body>
                <choice>
                    <op>
                        <!-- Receive reslts of extraction from OBSERVATORY Agent -->
                        <recv>
                            <mesg performative="inform-result">
                                <var name="opinion"/>
                                <var name="c_id"/>
                            </mesg>
                            <clause1>
                                <wild/>
                            </clause1>
                            <clause2>
                                <role name="observatory"/>
                            </clause2>
                        </recv>

                        <!-- Generate Combined Opinion -->
                        <proc name="GenerateOpinion">
                            <in>
                                <var name="opinion"/>
                            </in>
                            <out>
                                <var name="combined_opinion"/>
                            </out>
                        </proc>
                    </op>
                    <!-- OR Choice -->
                    <op>
                        <!-- Receive Failure Message -->
                        <recv>
                            <mesg performative="inform-failure">
                                <var name="c_id"/>
                            </mesg>
                            <clause1>
                                <wild/>
                            </clause1>
                        <clause2>
                            <role name="observatory"/>
                        </clause2>
                    </recv>

                    <!-- If Failed Execute the protocol again, this time
                         could be a different set of agents who take part  -->
                    <call name="contractnetsend">
                        <in>
                            <var name="unknown"/>
                        </in>
                    </call>
                </op>
            </choice>
        </body>
        <timeout>
            <null/>
        </timeout>
    </while>
</body>
</method>

<!-- CONTRACTREJECT Method -->
<method name="contractreject">
    <in>
        <var name="reject"/>
        <var name="c_id"/>
        <var name="accept"/>
        <var name="unknown"/>
    </in>
    <body>
        <choice>
            <op>
                <!-- Extract head of List -->
                <proc name="Head">
                    <in>
                        <var name="reject"/>
                    </in>
                    <out>
                        <var name="head"/>
                    </out>
                </proc>

                <!-- Extract tail of List -->
                <proc name="Tail">
                    <in>
                        <var name="reject"/>
                    </in>
                    <out>
                        <var name="tail"/>
                    </out>
                </proc>

                <!-- Send Reject-proposal to HEAD -->
                <send>
                    <mesg performative="reject-proposal">
                        <var name="c_id"/>
                    </mesg>
                    <clause1>
                        <var name="head"/>
                    </clause1>
                    <clause2>
                        <wild/>
                    </clause2>
                </send>

                <!-- Invoke CONTRACTREJECT Method -->
                <call name="contractreject">
                    <in>
                        <var name="tail"/>
                        <var name="c_id"/>
                        <var name="accept"/>
                        <var name="unknown"/>
                    </in>
                </call>
            </op>
            <!-- OR Choice -->
            <op>

                <call name="contractaccept">
                    <in>
                        <var name="accept"/>
                        <var name="unknown"/>
                        <var name="c_id"/>
                    </in>
                </call>
            </op>
        </choice>
    </body>
</method>
</agent>

<!-- CONTRACTNET Agent -->
<agent implementation="http://localhost:8080/MyContractService/contract?WSDL" max="1" min="1"
       role="contractnet">
    <method name="main">
        <body>
            <while>
                <body>
                    <!-- Receive request from CFP from ANY Agent -->
                    <recv>
                        <mesg performative="request">
                            <var name="potential_agents"/>
                            <var name="proposal"/>
                        </mesg>
                        <clause1>
                            <var name="init"/>
                        </clause1>
                        <clause2>
                            <wild/>
                        </clause2>
                    </recv>

                    <!-- Generate Unique ID -->
                    <proc name="GenerateID">
                        <out>
                            <var name="c_id"/>
                        </out>
                    </proc>

                    <!-- Initialise variables -->
                    <proc name="Initialise">
                        <out>
                            <var name="success"/>
                        </out>
                    </proc>

                    <!-- Call CFP Method -->
                    <call name="cfp">
                        <in>
                            <var name="potential_agents"/>
                            <var name="proposal"/>
                            <var name="init"/>
                            <var name="c_id"/>
                        </in>
                    </call>

                    <!-- Restart Agent -->
                    <call name="main"/>
                </body>
                <timeout>
                    <null/>
                </timeout>
            </while>
    </body>
</method>

<!-- CALL FOR PARTICIPATION (CFP) Method -->
<method name="cfp">
    <in>
        <var name="potential_agents"/>
        <var name="proposal"/>
        <var name="init"/>
        <var name="c_id"/>
    </in>
    <body>
        <choice>
            <op>
                <!-- Extract head of List -->
                <proc name="Head">
                    <in>
                        <var name="potential_agents"/>
                    </in>
                    <out>
                        <var name="head"/>
                    </out>
                </proc>

                <!-- Extract tail of List -->
                <proc name="Tail">
                    <in>
                        <var name="potential_agents"/>
                    </in>
                    <out>
                        <var name="tail"/>
                    </out>
                </proc>

                <!-- Send CFP to agent represented by $head -->
                <send>
                    <mesg performative="cfp">
                        <var name="proposal"/>
                        <var name="init"/>
                        <var name="c_id"/>
                    </mesg>
                    <clause1>
                        <var name="head"/>
                    </clause1>
                    <clause2>
                        <wild/>
                    </clause2>
                </send>

                <!-- Recursive call on tail -->
                <call name="cfp">
                    <in>
                        <var name="tail"/>
                        <var name="proposal"/>
                        <var name="init"/>
                        <var name="c_id"/>
                    </in>
                </call>
            </op>
            <!-- OR Choice -->
            <op>
                <call name="receiveproposals">
                    <in>
                        <var name="init"/>
                    </in>
                </call>
            </op>
        </choice>
    </body>
</method>

<!-- RECEIVE PROPOSAL Method -->
```

```
<method name="receiveproposals">
    <in>
        <var name="init"/>
    </in>
    <body>
        <while>
            <body>
                <choice>
                    <op>
                        <!-- Receive Propose message from OBSERVATORY Agent -->
                        <recv>
                            <mesg performative="propose">
                                <var name="c_id"/>
                            </mesg>
                            <clause1>
                                <var name="agent"/>
                            </clause1>
                            <clause2>
                                <wild/>
                            </clause2>
                        </recv>

                        <!-- Store the agents who issued ACCEPT -->
                        <proc name="StoreProposal">
                            <in>
                                <var name="agent"/>
                                <var name="c_id"/>
                            </in>
                        </proc>

                        <!-- Recursive call on RECEIVEPROPOSAL Method -->
                        <call name="receiveproposals">
                            <in>
                                <var name="init"/>
                            </in>
                        </call>

                    </op>
                    <!-- OR Choice -->
                    <op>
                        <!-- Receive Propose message from OBSERVATORY Agent -->
                        <recv>
                            <mesg performative="refuse">
                                <var name="c_id"/>
                            </mesg>
                            <clause1>
                                <var name="agent"/>
                            </clause1>
                            <clause2>
                                <wild/>
                            </clause2>
                        </recv>

                        <!-- Store the agents who issued REJECT -->
                        <proc name="StoreRefusal">
                            <in>
                                <var name="agent"/>
                                <var name="c_id"/>
                            </in>
                        </proc>

                        <!-- Recursive call on RECEIVEPROPOSAL Method -->
                        <call name="receiveproposals">
                            <in>
                                <var name="init"/>
                            </in>
                        </call>
                    </op>
                </choice>
            </body>
            <timeout>
                <!-- Retrieve open proposals -->
```

```
                    <proc name="RetrieveProposals">
                        <out>
                            <var name="open_proposals"/>
                        </out>
                    </proc>

                    <!-- Send all the agents who accepted the proposal back to the initiator -->
                    <send>
                        <mesg performative="response">
                            <var name="open_proposals"/>
                            <var name="c_id"/>
                        </mesg>
                        <clause1>
                            <var name="init"/>
                        </clause1>
                        <clause2>
                            <wild/>
                        </clause2>
                    </send>
                </timeout>
            </while>
        </body>
    </method>
</agent>

<!-- OBSERVATORY Agent -->
<agent implementation="http://localhost:8080/MyObservatoryService/observatory?WSDL" max="1" min="1"
       role="observatory">

    <!-- MAIN Method -->
    <method name="main">
        <body>
            <while>
                <body>
                    <!-- Receive CFP Request -->
                    <recv>
                        <mesg performative="cfp">
                            <var name="proposal"/>
                            <var name="init"/>
                            <var name="c_id"/>
                        </mesg>
                        <clause1>
                            <var name="contractnet"/>
                        </clause1>
                        <clause2>
                            <role name="contractnet"/>
                        </clause2>
                    </recv>

                    <!-- Initialise -->
                    <proc name="Initialise">
                        <out>
                            <var name="success"/>
                        </out>
                    </proc>
                </body>
                <timeout>
                    <null/>
                </timeout>
            </while>
            <choice>
                <op>
                    <!-- Consider Proposal (ACCEPT or REJECT) -->
                    <proc name="ConsiderProposal">
                        <in>
                            <var name="proposal"/>
                        </in>
                    </proc>

                    <!-- Send Propose message back to the CONTRACTNET Agent -->
                    <send>
                        <mesg performative="propose">
                        <var name="c_id"/>
                    </mesg>
                    <clause1>
                        <var name="contractnet"/>
                    </clause1>
                    <clause2>
                        <role name="contractnet"/>
                    </clause2>
                </send>

                <!-- Call WAITFORDECISION Method -->
                <call name="waitfordecision">
                    <in>
                        <var name="init"/>
                        <var name="proposal"/>
                    </in>
                </call>
            </op>
            <!-- OR Choice -->
            <op>
                <!-- Send refuse message back to the CONTRACTNET Agent -->
                <send>
                    <mesg performative="refuse">
                        <var name="c_id"/>
                    </mesg>
                    <clause1>
                        <var name="contractnet"/>
                    </clause1>
                    <clause2>
                        <role name="contractnet"/>
                    </clause2>
                </send>

                <!-- Restart Agent -->
                <call name="main"/>
            </op>
        </choice>
    </body>
</method>

<!-- WAITFORDECISION Method -->
<method name="waitfordecision">
    <in>
        <var name="init"/>
        <var name="proposal"/>
    </in>
    <body>
        <while tmax="30">
            <body>
                <choice>
                    <op>
                        <!-- Receive an accept-proposal frm the INITIATOR -->
                        <recv>
                            <mesg performative="accept-proposal">
                                <var name="c_id"/>
                            </mesg>
                            <clause1>
                                <var name="init"/>
                            </clause1>
                            <clause2>
                                <wild/>
                            </clause2>
                        </recv>

                        <!-- If necessary divide the proposal -->
                        <proc name="DivideProposal">
                            <in>
                                <var name="proposal"/>
                            </in>
                            <out>
                                <var name="proposal_sections"/>
                            </out>
                        </proc>

                        <!-- Call EXTRACT Method -->
                        <call name="extract">
                            <in>
                                <var name="proposal_sections"/>
                                <var name="init"/>
                                <var name="c_id"/>
                            </in>
                        </call>
                    </op>
                    <!-- OR Choice -->
                    <op>
                        <!-- Receive an reject-proposal frm the INITIATOR -->
                        <recv>
                            <mesg performative="reject-proposal">
                                <var name="c_id"/>
                            </mesg>
                            <clause1>
                                <var name="init"/>
                            </clause1>
                            <clause2>
                                <wild/>
                            </clause2>
                        </recv>
                    </op>
                </choice>
            </body>
            <timeout>
                <null/>
            </timeout>
        </while>
    </body>
</method>

<!-- EXTRACT Method -->
<method name="extract">
    <in>
        <var name="proposal_sections"/>
        <var name="init"/>
        <var name="c_id"/>
    </in>
    <body>
        <choice>
            <op>
                <!-- Extract head of List -->
                <proc name="Head">
                    <in>
                        <var name="proposal_sections"/>
                    </in>
                    <out>
                        <var name="head"/>
                    </out>
                </proc>

                <!-- Extract tail of List -->
                <proc name="Tail">
                    <in>
                        <var name="proposal_sections"/>
                    </in>
                    <out>
                        <var name="tail"/>
                    </out>
                </proc>

                <!-- Send Request to any EXTRACTION Agent -->
                <send>
                    <mesg performative="request-extraction">
                        <var name="head"/>
                    </mesg>
                    <clause1>
                        <wild/>
                    </clause1>
                    <clause2>
                        <role name="extraction"/>
                    </clause2>
                </send>

                <!-- Call EXTRACT Method -->
                <call name="extract">
                    <in>
                        <var name="tail"/>
                        <var name="init"/>
                        <var name="c_id"/>
                    </in>
                </call>
            </op>
            <!-- OR Choice -->
            <op>
                <call name="wait">
                    <in>
                        <var name="init"/>
                        <var name="c_id"/>
                    </in>
                </call>
            </op>
        </choice>
    </body>
</method>

<!-- WAIT Method -->
<method name="wait">
    <in>
        <var name="init"/>
        <var name="c_id"/>
    </in>
    <body>
        <choice>
            <op>
                <!-- Finished Check -->
                <proc name="Finished">
                    <in>
                        <var name="c_id"/>
                    </in>
                    <out>
                        <var name="success"/>
                    </out>
                </proc>

                <while tmax="30">
                    <body>

                        <!-- Receive response from EXTRACTION Agent -->
                        <recv>
                            <mesg performative="response-extraction">
                                <var name="data"/>
                            </mesg>
                            <clause1>
                                <wild/>
                            </clause1>
                            <clause2>
                                <role name="extraction"/>
                            </clause2>
                        </recv>

                        <!-- Store the Result -->
                        <proc name="Store">
                            <in>
                                <var name="data"/>
                            </in>
                        </proc>

                        <!-- One less agent to receive -->
                        <proc name="Remove">
```

```
                            <out>
                                <var name="success"/>
                            </out>
                        </proc>

                        <!-- Recursvie Call -->
                        <call name="wait">
                            <in>
                                <var name="init"/>
                                <var name="c_id"/>
                            </in>
                        </call>
                    </body>
                    <timeout>
                        <!-- Send inform-failure if the agent has been waiting too long  -->
                        <send>
                            <mesg performative="inform-failure">
                                <var name="c_id"/>
                            </mesg>
                            <clause1>
                                <var name="init"/>
                            </clause1>
                            <clause2>
                                <wild/>
                            </clause2>
                        </send>
                        <null/>
                    </timeout>
                </while>
            </op>
            <!-- OR Choice -->
            <op>
                <call name="finish">
                    <in>
                        <var name="init"/>
                        <var name="c_id"/>
                    </in>
                </call>
            </op>
        </choice>
    </body>
</method>

<!-- FINISH Method -->
<method name="finish">
    <in>
        <var name="init"/>
        <var name="c_id"/>
    </in>
    <body>
        <!-- Derive opinion -->
        <proc name="ExtractData">
            <out>
                <var name="opinion"/>
            </out>
        </proc>

        <!-- Send final results back to the INITIATOR -->
        <send>
            <mesg performative="inform-result">
                <var name="opinion"/>
                <var name="c_id"/>
            </mesg>
            <clause1>
                <var name="init"/>
            </clause1>
            <clause2>
                <wild/>
            </clause2>
        </send>
    </body>
</method>
```

```
</agent>

<!-- EXTRACTION Agent -->
<agent implementation="http://localhost:8080/MyExtractService/extract?WSDL"
       max="1" min="1"
       role="extraction">
    <!-- MAIN Method -->
    <method name="main">
        <body>
            <while tmax="50">
                <body>
                    <!-- Receive Coordination Request from OBSERVATORY Agent -->
                    <recv>
                        <mesg performative="request-extraction">
                            <var name="proposal_section"/>
                        </mesg>
                        <clause1>
                            <var name="coordinator"/>
                        </clause1>
                        <clause2>
                            <wild/>
                        </clause2>
                    </recv>

                    <!-- Initialise variables -->
                    <proc name="Initialise">
                        <out>
                            <var name="success"/>
                        </out>
                    </proc>

                    <!-- Call RETRIEVE Method -->
                    <call name="retrieve">
                        <in>
                            <var name="proposal_section"/>
                            <var name="coordinator"/>
                        </in>
                    </call>

                    <!-- Restart Agent -->
                    <call name="main"/>
                </body>
                <timeout>
                    <null/>
                </timeout>
            </while>
        </body>
    </method>

    <!-- RETRIEVE Method -->
    <method name="retrieve">
        <in>
            <var name="proposal_section"/>
            <var name="coordinator"/>
        </in>
        <body>
            <choice>
                <op>
                    <!-- Extract head of List -->
                    <proc name="Head">
                        <in>
                            <var name="proposal_section"/>
                        </in>
                        <out>
                            <var name="head"/>
                        </out>
                    </proc>

                    <!-- Extract tail of List -->
                    <proc name="Tail">
                        <in>
                            <var name="proposal_section"/>
            </in>
            <out>
                <var name="tail"/>
            </out>
        </proc>

        <!-- Extract Service Definition -->
        <proc name="ExtractService">
            <in>
                <var name="head"/>
            </in>
            <out>
                <var name="service_def"/>
            </out>
        </proc>

        <!-- Extract Input Params -->
        <proc name="ExtractInput">
            <in>
                <var name="head"/>
            </in>
            <out>
                <var name="input_params"/>
            </out>
        </proc>

        <!-- Extract Output Params -->
        <proc name="ExtractOutput">
            <in>
                <var name="head"/>
            </in>
            <out>
                <var name="output_params"/>
            </out>
        </proc>

        <!-- Store the results of the Service Invocation -->
        <proc name="Store">
            <in>
                <var name="output_params"/>
            </in>
        </proc>

        <!-- Recursive Call -->
        <call name="retrieve">
            <in>
                <var name="tail"/>
                <var name="coordinator"/>
            </in>
        </call>
    </op>
    <!-- OR Choice -->
    <op>
        <!-- Retrieve data -->
        <proc name="Retrieve">
            <out>
                <var name="data"/>
            </out>
        </proc>

        <!-- Send results back to the COORDINATOR Agent -->
        <send>
            <mesg performative="response-extraction">
                <var name="data"/>
            </mesg>
            <clause1>
                <var name="coordinator"/>
            </clause1>
            <clause2>
                <wild/>
            </clause2>
        </send>
```

```
                        </op>
                    </choice>
                </body>
            </method>
        </agent>

        <!-- USER Agent -->
        <agent implementation="http://localhost:8080/MyRsmService/rsm?WSDL" max="1" min="1" role="user">
            <!-- MAIN Method -->
            <method name="main">
                <body>
                    <!-- Get unknown -->
                    <proc name="sayRaDec">
                        <out>
                            <var name="unknown"/>
                        </out>
                    </proc>

                    <!-- Send RA and DEC to RSM Agent -->
                    <send>
                        <mesg performative="request">
                            <var name="unknown"/>
                        </mesg>
                        <clause1>
                            <wild/>
                        </clause1>
                        <clause2>
                            <role name="classification"/>
                        </clause2>
                    </send>
                </body>
            </method>
        </agent>

    </Scene>
    <!-- Automated Scene -->
    <Scene name="automated">
        <!-- Port Definitions -->
        <output>
            <port name="automated_out1" type="xsd:string" core="true"/>
        </output>

        <!-- Simple Agent -->
        <agent implementation="http://localhost:8080/MyRsmService/auto?WSDL" max="1" min="1" role="automated">
            <method name="main">
                <body>
                    <!-- Get unknown -->
                    <proc name="sayRaDec">
                        <out>
                            <var name="unknown"/>
                        </out>
                    </proc>

                    <!-- Port Write -->
                    <portwrite name="automated_out1">
                        <var name="unknown"/>
                        <var name="unknown"/>
                    </portwrite>
                </body>
            </method>
        </agent>
    </Scene>
</sceneset>

<!-- Parameterisation -->
<mapping name="demomapping">

    <!-- LSST Scene -->
    <node location="" name="lsst">
        <role name="classification">
            <agent implementation="http://localhost:8080/MyReactiveService/reactive?WSDL" name="myCLASSIFY" num="1"
                   portwait="10"
                recvwait="10"/>
    </role>

    <role name="user">
        <agent implementation="http://localhost:8080/MyRsmService/rsm?WSDL" name="myUSER" num="1" portwait="10"
                recvwait="10"/>
    </role>

    <role name="contractnet">
        <agent implementation="http://localhost:8080/MyContractService/contract?WSDL" name="myCN" num="1"
                portwait="10"
                recvwait="10"/>
    </role>

    <role name="observatory">
        <agent implementation="http://localhost:8080/MyObservatoryService/observatory?WSDL" name="london"
                num="1"
                portwait="10"
                recvwait="10"/>
    </role>

    <role name="observatory">
        <agent implementation="http://localhost:8080/MyObservatory2Service/observatory2?WSDL" name="newyork"
                num="1"
                portwait="10"
                recvwait="10"/>
    </role>

    <role name="observatory">
        <agent implementation="http://localhost:8080/MyObservatory3Service/observatory3?WSDL" name="paris"
                num="1"
                portwait="10"
                recvwait="10"/>
    </role>

    <role name="observatory">
        <agent implementation="http://localhost:8080/MyObservatory4Service/observatory4?WSDL" name="edinburgh"
                num="1"
                portwait="10"
                recvwait="10"/>
    </role>

    <role name="extraction">
        <agent implementation="http://localhost:8080/MyExtractService/extract?WSDL"
                name="myExtraction"
                num="1"
                portwait="10"
                recvwait="10"/>
    </role>

    <role name="extraction">
        <agent implementation="http://localhost:8080/MyExtract2Service/extract2?WSDL"
                name="myExtraction2"
                num="1"
                portwait="10"
                recvwait="10"/>
    </role>
</node>

<!-- Automated Scene -->
<node location="" name="automated">

    <role name="automated">
        <agent implementation="http://localhost:8080/MyRsmService/rsm?WSDL"
                name="myAutomated"
                num="1"
                portwait="10"
                recvwait="10"/>
    </role>
</node>

<!-- Link Definition -->
```

```
        <link>
            <source>
                <outport port="automated_out1" scene="automated"/>
            </source>
            <sink>
                <inport port="lsst_in1" scene="lsst"/>
            </sink>
        </link>
    </mapping>
</protocol>
```

Bibliography

[1] Fipa ACL Message Structure Specification. Technical report, Foundation for Intelligent Physical Agents, December 2002.

[2] *Grid Computing: Making the Global Infrastructure a Reality*, pages 65–100. Wiley Series in Communication Networking and Distributed Systems, 2003.

[3] Web Services Coordination (WS-coordination). Technical report, BEA Systems and IBM and Microsoft Corporation, September 2003.

[4] P. Allan, B. Bentley, C. Davenhall, S. Garrington, D. Giaretta, L. Harra, M. Irwin, A. Lawrence, M. Lockwood, B. Mann, R. McMahon, F. Murtagh, J. Osborne, C. Page, C. Perry, D. Pike, A. Richards, G. Rixon, J. Sherman, R. Stamper, and M. Watson. Astrogrid. Technical report, Available at: `www.astrogrid.org`, 2001.

[5] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, André Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. Enabling Applications on the Grid: A GridLab Overview. *International Journal of High Performance Computing Applications: Special Issue on Grid Computing: Infrastructure and Applications*, 17(4):449–466, November 2003.

[6] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludaescher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th International Conference on Scientific and Statistical Database Management (SSDBM'04), Santorini Island, Greece,* June 2004.

[7] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Lan-

guage for Web Services Specification, Version 1.1. Technical report, BEA Systems and IBM Corporation and Microsoft Corporation and SAP AG and Siebel Systems, May 2003.

[8] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.

[9] J.L. Austin. How to Do Things with Words. *Oxford University Press*, 1962.

[10] Adam Barker and Robert G. Mann. Cooperative Information Agents X, 10th International Workshop, CIA 2006, Edinburgh, UK, September 11-13, 2006, proceedings. In Matthias Klusch, Michael Rovatsos, and Terry R. Payne, editors, *CIA*, volume 4149 of *Lecture Notes in Computer Science*, pages 446–460. Springer, 2006.

[11] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[12] E. Bertin and S. Arnouts. Sextractor: Software for source extraction, Astronomy and Astrophysics, Suppl. Ser., 117:393–404, June 1996.

[13] Micol Bolzonella, Roser Pell, and Joan-Marc Miralles. *HyperZ, v:1.1, User's Manual*.

[14] S. Bowers and B. Ludaescher. Actor-Oriented Design of Scientific Workflows. In *Lecture Notes in Computer Science Volume 3716*, pages 369–384. Springer Berlin / Heidelberg, June 2005.

[15] Doug Bunting, Martin Chapman, Oisin Hurley, Mark Little, Jeff Mischkinsky, Eric Newcomer, Jim Webber, and Keith Swenson. Web Services Composite Application Framework. Technical report, Arjuna Technologies Ltd and Fujitsu Limited and IONA Technologies Ltd and Oracle Cooperation and Sun Microsystems, July 2003.

[16] B. Cavanagh, A. Allan, T. Jenness, F. Economou, P. Hirst, A. Adamson, and T. Naylor. Architecture of the WFCAM/eSTAR Transient Object Detection Agent. In *Astronomical Society of the Pacific Conference Series*, pages 504–+, December 2005.

[17] Adrian A. Collister and Ofer Lahav. Annz: estimating photometric redshifts using artificial neural networks. *Publications of the Astronomical Society of the Pacific*, 16:345, 2004.

[18] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design, Fourth Edition*, pages 565–599. Addison Wesley, 2005.

[19] Karl Czajkowski, Donald F Ferguson, Ian Foster, Jeffrey Frey, Steve Graham, Igor Sedukhin, David Snelling, Steve Tuecke, and William Vambenepe. The Ws-Resource Framework, Version 1.0. Technical report, Globus, May 2004.

[20] Scientific Data Management Framework Workshop. Argonne National Labs. `http://sdm.lbl.gov/sdmcenter`, August 2003.

[21] e-Science Grid Environments Workshop. e-Science Institute, Edinburgh, Scotland. `http://www.nesc.ac.uk/esi/events`, May 2004.

[22] Scotland e-Science Institute, Edinburgh. Astrogrid Deployment and Development Workshop. `http://www.nesc.ac.uk/esi/events/646/`, January 2006.

[23] e-Science Workflow Services Workshop. e-Science Institute, Edinburgh, Scotland. `http://www.nesc.ac.uk/esi/events/303`, December 2003.

[24] C.A. Ellis and J. G. Nutt. *Office Information Systems and Computer Science*, pages 27–60. ACM Computing Surveys, 1980.

[25] Greg Meredith Erik Christensen, Francisco Curbera and Sanjiva Weerawarana. *Web Services Desription Language (WSDL) Specification 1.1*. World Wide Web Consortium (W3C), March 2001.

[26] M. Esteva, J. Rodriguez, J. Arcos, C. Sierra, and P. Garcia. Formalising Agent Mediated Electronic Institutions. In *Catalan Congres on AI (CCIA'00)*, pages 29–38, 2000.

[27] David C. Fallside and Priscilla Walmsley. Xml Schema Part 0: Primer Second Edition. Technical report, World Wide Web Consoritum (W3C), 2004.

[28] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, volume LNCS 3779, pages 2–13. Springer-Verlag, 2005.

[29] I. Foster, N. R. Jennings, and C. Kesselman. Brain meets Brawn: Why Grid and Agents Need Each Other. In *Proc. 3rd Int. Conf. on Autonomous Agents and Multi-Agent Systems*, New York, USA, 2004.

[30] James Hendler. Science and the Semantic Web. In *Science*, pages 520–521. January 2003.

[31] David Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition, Document Number tc00-1003 edition, January 1995.

[32] The UKIRT Infrared Deep Sky Survey:
`http://www.ukidss.org`.

[33] Y. Labrou J. Mayiield and T. Finin. Evaluating kqml as an Agent Communication Language. In *Intelligent Agents I[(L AI Volume I037)*, pages 347–360, Berlin, Germany, 1996. Springer-Verlag.

[34] Nicholas R. Jennings. Agent-Oriented Software Engineering. In Francisco J. Garijo and Magnus Boman, editors, *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)*, volume 1647, pages 1–7. Springer-Verlag: Heidelberg, Germany, 30– 2 1999.

[35] J.Zhao, C.A. Goble, M. Greenwood, C. Wroe, and R. Stevens. Annotating, linking and browsing provenance logs for e-science. In *1st Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data, Sanibel Island, Florida, USA*, October 2003.

[36] Ananth Krishna, Victor Tan, Richard Lawley, Simon Miles, and Luc Moreau. The mygrid notification service. In *Proceedings of The UK OST e-Science second All Hands Meeting*, pages 475–482, 2003.

[37] D. Lambert and D. Robertson. Matchmaking and brokering multi-party interactions using historical performance data. In *Fourth International Joint Conference on Autonomous Agents and Multi Agent Systems*. Springer, 2005.

[38] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, Special Issue on Scientific Workflows, 2005.

[39] David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burnstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry R. Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara. Bringing Semantics to Web Services: The OWL-S Approach. 2004.

[40] Noah Mendelsohn Jean-Jacques Moreau Martin Gudgin, Marc Hadley and Henrik Frystyk Nielsen. *Simple Object Access Protocol (SOAP) 1.2 Specification*. World Wide Web Consortium (W3C), June 2003.

[41] A. Mayer, S. McGough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, and J. Darlington. Meaning and Behaviour in Grid Oriented Components. In *Lecture Notes in Computer Science*, volume 2536, pages 100–111. Springer-Verlag Berlin Heidelberg, 2002.

[42] William K. Michener. Building SEEK: The Science Environment for Ecological Knowledge. *DataBits: An electronic newsletter for Information Managers*, Spring Edition, 2003.

[43] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A tool for the Composition and Enactment of Bioinformatics Workflows. In *Bioinformatics Journal 20(17)*, pages 3045–3054, 2004.

[44] Tom Oinn, Mark Greenwood, Carole Goble, Matthew Addis, Justin Ferris, Darren Marvin, Anil Wipat, Peter Li, and Tim Carver. Delivering web service coordination capability to users. In *Thirteenth International World Wide Web Conference (WWW2004), New York*, pages 438–439, 2004.

[45] Ed Ort. Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools. Technical report, Sun Microsystems, April 2004.

[46] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[47] Elliotte Rusty Harold. *Processing XML with Java*. Addison Wesley, 2002.

[48] Shazia W. Sadiq, Wasim Sadiq, and Maria E. Orlowska. Pockets of flexibility in workflow specification. In *ER ’01: Proceedings of the 20th International Conference on Conceptual Modeling*, pages 513–526, London, UK, 2001. Springer-Verlag.

[49] Inderjeet Singh, Beth Stearns, Beth Stearns, Sean Brydon, and Greg Murray. *Designing Web Services with the J2EE(TM) 1.4 Platform : JAX-RPC, SOAP, and XML Technologies*. Pearson Education, 2004.

[50] R. Smith. The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.

[51] Sebastian Stein, Nicholas R. Jennings, and Terry R. Payne. Flexible provisioning of service workflows. In *ECAI*, pages 295–299, 2006.

[52] Robert Stevens, Kevin Glover, Chris Greenhalgh, Claire Jennings, Simon Pearce, Melena Radenkovic, and Anil Wipat. Performing in-silico Experiments on the Grid: A Users Perspective. In *Proceedings of the UK e-Science all hands meeting*, September 2003.

[53] Robert Stevens, Robin McEntire, Carole Goble, Mark Greenwood, Jun Zhao, Anil Wipat, and Peter Li. $^{\mathrm{my}}$Grid and the Drug Discovery Process. *Drug Discovery Today: BIOSILICO*, 4(2):140–148, 2004.

[54] Ian J. Taylor, Matthew S. Shields, Ian Wang, and Roger Philp. Distributed P2P Computing within Triana: A Galaxy Visualization Test Case. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, pages 16–27. IEEE Computer Society, 2003.

[55] Ptolemy II Project:
`http://ptolemy.eecs.berkeley.edu/ptolemyII`.

[56] Large Synoptic Survey Telescope (LSST):
`http://www.lsst.org`.

[57] OpenKnowledge Project:
`http://www.openk.org`.

[58] Proteus Technologues:
`http://www.proteus-technologies.com`.

[59] The Wide Field Astronomy Unit (WFAU):
`http://www.roe.ac.uk/ifa/wfau`.

[60] Sloan Digital Sky Survey:
`http://www.sdss.org`.

[61] The Unified Modelling Language:
`http://www.uml.org`.

[62] The XMM-Newton Science Archive:
`http://xmm.vilspa.esa.es/xsa`.

[63] C. M. Sperberg-McQueen Eve Maler Tim Bray, Jean Paoli and Franois Yergeau. *Extensible Markup Language (XML) 1.0 Specification (Fourth Edition)*. World Wide Web Consortium (W3C), August 2006.

[64] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open Grid Services Infrastructure (ogsi), Version 1.0. Technical report, Global Grid Forum (GGF), June 2003.

[65] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. In *Distributed and Parallel Databases*, pages 5–51, July 2003.

[66] International Virtual Observatory Alliance:
`http://www.ivoa.net`.

[67] Gottfried Vossen and Mathias Weske. The wasa2 object-oriented workflow management system. *SIGMOD Rec.*, 28(2):587–589, 1999.

[68] Chris Walton. Dialogue Protocols for Multi-Agent Systems. Technical report, University of Edinburgh, September 2003.

[69] Christopher D. Walton and Adam D. Barker. An Agent-Based e-Science Experiment Builder. In *Semantic Intelligent Middleware for the Web and the Grid*. ECAI, 2004.

[70] Web Services Description Language for Java (WSDL4J):
`http://sourceforge.net/projects/wsdl4j`.

[71] Mathias Weske, Gottfried Vossen, and Claudia Bauzer Medeiros. Scientific Workflow Management: WASA Architecture and Applications. Technical report, University of Muenster and University of Campinas, January 1996.

[72] Alexander E. Wise, Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, and Stanley M. Sutton. Using little-JIL to coordinate agents

in software engineering. In *Automated Software Engineering*, pages 155–164, 2000.

[73] Michael Wooldrige. *An Introduction to Multiagent Systems*, pages 27–60. John Wiley and Sons Ltd, 2002.

[74] FreeFluo Workflow Enactment Engine: `http://freefluo.sorceforge.net`.

[75] GRIST Workshop on Service Composition for Data Exploration in the Virtual Observatory. California Institute of Technology. `http://grist.caltech.edu/sc4devo`, July 2004.

[76] Jun Yan. *A Framework and Coordination Technologies for Peer-to-peer Based Decentralised Workflow Systems*. PhD thesis, School of Information Technology, Swinburne University of Technology, August 2004.

[77] Jun Zhao, Robert Stevens, Chris Wroe, Mark Greenwood, and Carole Goble. The origin and history of in-silico experiments. In *Proceedings of the UK e-Science All Hands Meeting, Nottingham UK*, September 2004.