Flexible Service Composition

Adam Barker[1] and Robert G. Mann[2]

[1] Centre for Intelligent Systems and their Applications (CISA)
School of Informatics, University of Edinburgh, UK
Email: a.d.barker@ed.ac.uk
[2] Institute for Astronomy, University of Edinburgh, UK

Abstract. Both the agent and Grid communities develop concepts for distributed computing, however they do so with different motivations. This paper demonstrates how the flexible coordination technique of interaction protocols, from the field of multiagent communication, can be used to model the processes found in scientific workflow, a typical composition problem faced by the Grid community. Our approach is founded on the adaptation of the MultiAgent Protocol (MAP) language to perform web service composition. A definition of the language and framework is presented in order to solve a detailed scientific workflow, taken from the field of time-domain astronomy. MAP offers a flexible, adaptable approach, allowing 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.

1 Introduction

Scientists are increasingly 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 such as bio-informatics, geo-informatics and neuro-informatics [10]. The term *'Grid'* refers to the 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* [4]. The Grid is the machinery which enables e-Science. 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.

Both the agent and Grid communities develop concepts for distributed computing, however they do so from differing points of view. The agent community's focus lies with creating autonomous, flexible software components. Agents are designed to operate in dynamic and uncertain environments, making decisions at run-time. Communities of agents exhibit flexible cooperation and coordination through techniques such argumentation and social laws. The Grid community however, has focused on the development of middleware, which provides reliable, scalable and secure access to distributed resources. It is clear that these

two communities of research are starting to see a convergence of interests. The typical features of each community are illustrated by figure 1. In practise however, the application of techniques from the multiagent systems community to the Grid is a relatively new research area, as highlighted in [7]. Although the field is starting to see an increased level of interest, demonstrated by the recent series of workshops [3], [2] and new journal publication [9].

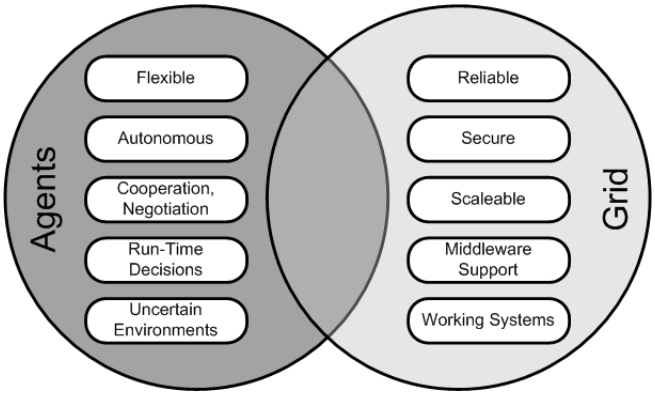

Fig. 1. A Convergence of Interests

The research presented in this paper addresses the problem of composing multiple services to form an e-Science experiment, or workflow [8]. There are a plethora of organisations creating Business Process Modelling languages. The current front runner is BPEL (Business Process Execution Language) [1] for web services, but there are many competing standards which occupy the same space [16]. Although scientific and business workflows have an overlapping set of requirements, it is also true that they each have their own domain specific requirements, and therefore need consideration separately. There are however, very few languages which deal with the flexible *knowledge acquisition* and *discovery processes* found in the sciences. Kepler [5], ICENI [11] and myGrid [14] are the current state of the art in scientific workflow composition, all using a dataflow modelling paradigm in order to capture the series of steps required to describe a distributed e-Science experiment.

This paper aims to demonstrate how the flexible coordination technique of interaction protocols, from the field of multiagent communication, can be used to model the processes found in scientific workflow, a problem from the Grid community. Allowing 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.

The remainder of this paper is structured as follows. Sections 2 and 3 introduce a motivating scientific workflow, taken from the Large Synoptic Survey

Telescope (LSST). This scenario demonstrates the need for of agent-based techniques, as the systems which perform this computation need to be reactive, collaborative and flexible systems. In section 4 a proposed framework and interaction protocol language is discussed, as a way to address the requirements laid down by the scenario. This language and framework is then applied to the motivating scenario in section 5, demonstrating the use of interaction protocols to model scientific workflow. Conclusions and current implementation work are then discussed in section 6.

2 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. 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. 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.

2.1 Change in the Universe

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 vision 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's going on in the universe.

3 Time-Domain Astronomy Scenario

The Large Synoptic Survey Telescope (LSST) [15] has been proposed to address many of these difficulties and open up 'time domain' astronomy, the telescope

will be able to tile the entire night sky over a three night period, generating 36 gigabytes of data every 30 seconds. This section introduces a motivating scenario taken from the LSST science use cases, an influential 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 then be queried instantly, providing a different perspective on the event. As data is collected it will be added to all the data previously detected from the same location of sky to create a very deep *master image*.

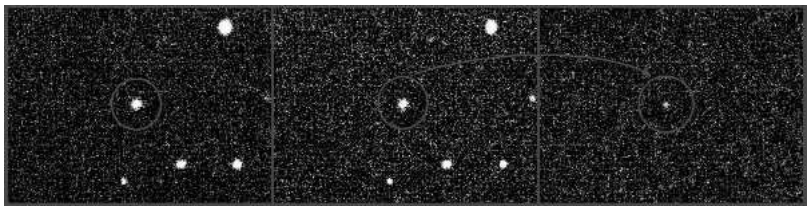

Fig. 2. An example of a Subtracted Image

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 2 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. The first task involves computing 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. A query is made to the *orbit catalogue*, which contains data about all known orbits. The results of this query are then cross matched with the subtracted image, leaving only objects which cannot be classified, and hence may be a new object discovery, or orbit. Further processing is performed, to try and compute smaller sections of orbit, known as a *tracklets*. If these smaller sections of orbits can be extrapolated (by cross referencing them with observations at earlier points in time), these new orbits, along with re-detections of known objects are updated in the orbit catalogue. With each re-detection of a known object, more information is provided, increasing the accuracy and further constraining the orbit. This process allows an accurate map of the sky to be built up, catching transient events in the act.

3.1 An Agent-Oriented Approach

The classification process described in section 3 is for known classes of object, but the hope is that, since LSST will provide a first attempt at time domain astronomy, it will discover new classes of object, previously undetected. Once the initial processing has finished, there will be some data which is left over. This data includes objects and orbits which can't be classified by the processing software. Typically, most of these objects will simply be junk, but this may only be revealed on the basis of comparison with other detections made from the same night. The systems which attempt to classify this data need to be reactive, collaborative, intelligent systems. On this basis, agent based techniques have been applied to the classification problem. It is important to note that certain details are left intentionally abstract, the moving objects scenario serves as a motivating factor, illustrating the kinds of features that our interaction protocol language and framework are required to model.

It is intended that agents will take over where the subtracted image processing left off. Groups of agents form a multiagent system, working on behalf of an observatory, in an attempt to classify whatever data is left over from the automated processing stage. Agents are initially set up with a certain amount of knowledge about properties of the data, and a number of statistical tests to perform. Agents need to cooperate and coordinate with one another, hence they are also set up with some rules about when and how to share information. Engineers can focus on developing individual, intelligent agents which are specialised in their own right. For example certain agents will have expertise on pixel failures on the camera, others contain data and a hypothesis about a certain kind of unclassified object. Figure 3 is 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 agents (illustrated by the square). Web services are shown as rounded rectangles. Communication between agents is shown by arrowed solid lines, web service invocations are shown as single arrowed dotted lines. An example interaction between a group of agents could be viewed as the following.

Agents at observatory A are attempting to classify objects left over from the image processing, one of the agents 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 agent 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. 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 [13] is executed over a group of observatory agents known to have possible data about the area of sky we are interested in, at time t. This is

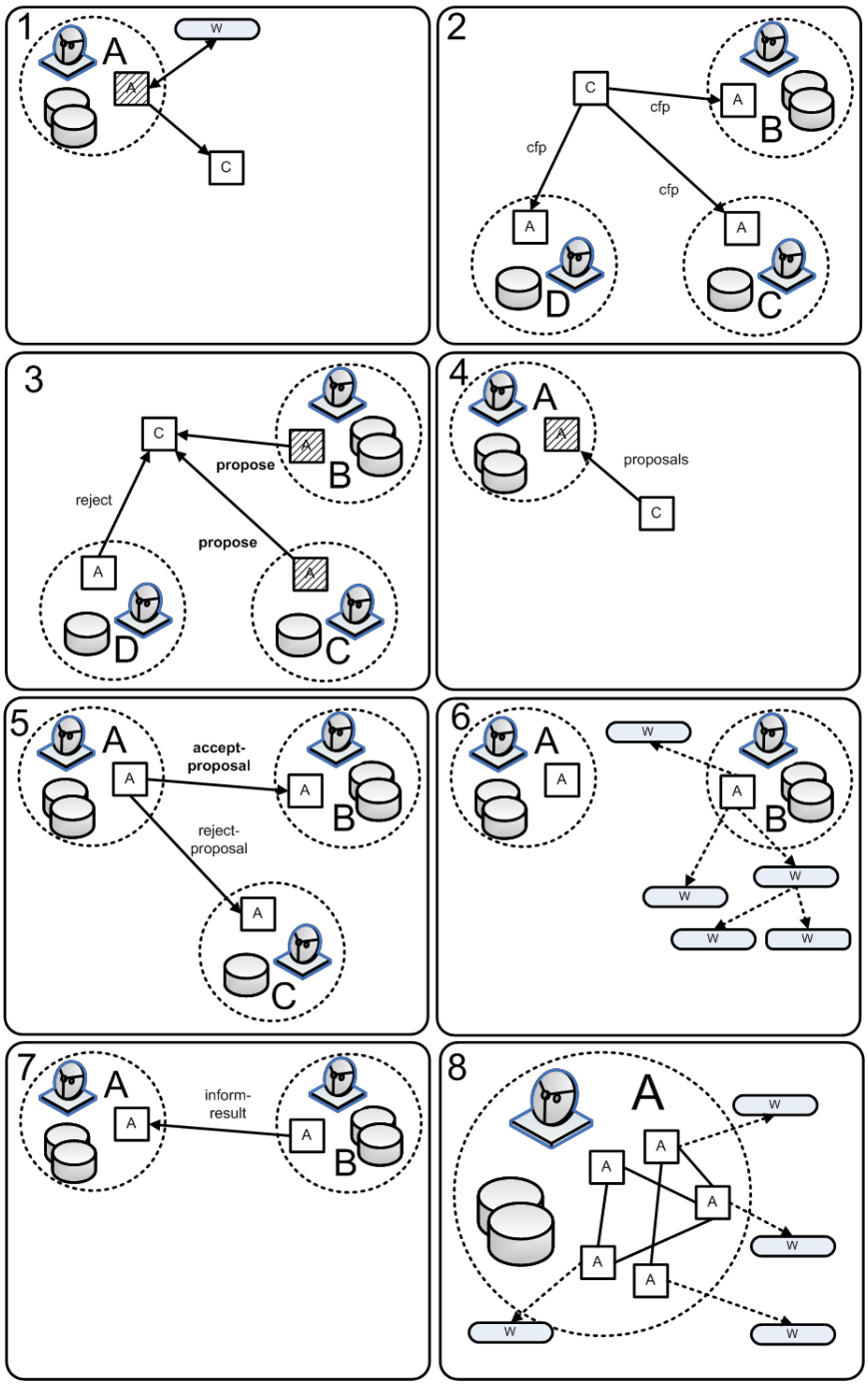

Fig. 3. Overview of LSST Scenario

illustrated by steps 1 to 4 of figure 3. A contract net agent (on behalf of the observatory) issues a call for participation over the set of possible observatory agents. The call for participation contains a proposal, defining the terms of agreement. The observatory agents then communicate within their local multiagent system to try and reach some form of conclusion about participation, issuing either an accept or reject message to the proposal. The set of agents who returned accept (in this case observatories B and C) are returned to the classification agent, who locally decides (based on some internal local knowledge and runtime conditions) which agent to obtain the data from. Step 5 of figure 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 agent to locally retrieve and process the data in accordance to the agreed contract net proposal (step 6 of figure 3), this will involve negotiation of agents local to observatory B and a set of external web service calls. Once this process has finished, the data is sent back to observatory A. Here the agents 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. Agents then continue to process the remainder of the junk data, following the same process again if an object cannot be classified locally. The paper now proposes an Agent Coordination Framework to address the problem of communication and web service invocation by agents in a distributed open, environment in order to solve the scenario detailed in this section.

4 Agent Coordination Framework

Multi Agent Protocols or MAP for short is an *interaction protocol* [12]. An interaction protocol is essentially a collection of conventions which allow agents in an open multiagent system to interact with one another. The term *open multiagent* 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.

The work of the MAP language builds upon the foundations laid down by the Electronic Institutions [6] framework; a popular technique for providing structure and organisation in an open multiagent system. It is designed as a light weight language to coordinate agents in an open multiagent system. Being lightweight it is therefore relatively sparse in features, however more complex semantics can, if required be layered on top of the basic MAP language. The abstract syntax of the MAP language is shown in figure 4.

The division of agent interactions into *scenes* is a key concept in the MAP language. Scenes can be thought of as a bounded space in which a group of agents interact on a single shared task. Scenes also allow the division of a large and complex protocol to be broken up into more manageable chunks. Scenes allow a measure of security to be places on a protocol, allowing agents which are not relevant to the protocol to be excluded from the scene. The most basic component

in this framework is an agent, which is defined by a unique name:`n` and a role:`r`. The *role* of an agent is fixed until the end of the scene and 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. Roles also allow us to specify multicast communication in MAP. For example, we can broadcast messages to all agents of a specific role.

An Agent's behaviour is defined by a set of Methods {`M`}, which can optionally take a list of Terms as arguments $\phi^{(k)}$. Methods are constructed from an Operation Set `op`, which enforce control flow in the agent and a set of actions α, which allow the agent to communicate and interact with a reasoning layer.

$P \in$ Protocol	$::= n\ (r\{M\})^{+}$	(Protocol)
$M \in$ Method	$::= \text{method}(\phi^{(k)}) = \text{op}$	(Method)
$op \in$ Operation	$::= \alpha$	(Action)
	$\mid\ op_1$ `then` op_2	(Sequence)
	$\mid\ op_1$ `or` op_2	(Choice)
	$\mid\ op_1$ `par` op_2	(Parallel Composition)
	$\mid$ `waitfor` op_1 `timeout` op_2	(Iteration)
	$\mid$ `invoke`$(\phi^{(k)})$	(Recursion)
$\alpha \in$ Action	$::= \epsilon$	(No Action)
	$\mid\ \phi^{(k)} = p(\phi^{(l)})$ fault $\phi^{(m)}$	(Decision Procedure)
	$\mid\ p(\phi^{(k)})$ `=>` `agent`$(\phi^{(1)}, \phi^{(2)})$	(Send)
	$\mid\ p(\phi^{(k)})$ `<=` `agent`$(\phi^{(1)}, \phi^{(2)})$	(Receive)
$\phi \in$ Term	$::= v \mid a \mid r \mid c \mid \_$	(Terms)

Fig. 4. MAP Abstract Syntax.

Actions α, can have side-effects and fail. Failure of actions causes backtracking of the protocol. The action set firstly consists of the decision procedure. The decision procedure set is implemented as a set of methods, exposed as a *reasoning web service*. When an agent needs to make an internal decision, it invokes methods on this web service; for example the logic deciding which observatory agent to choose after the initial round of the contract net protocol. Given a list of input Terms $\phi^{(l)}$, a procedure will invoke the required method on the reasoning web service `p`, using the terms as input. If required it will produce a list of output terms $\phi^{(k)}$ (results from the procedure) which can be referenced throughout the duration of the agents execution cycle. A procedure can raise an exception, in which case the fault terms $\phi^{(m)}$ are bound to the exception parameters and backtracking of the protocol occurs.

The remaining two actions that an agent can reference are the send and receive actions. Interaction between the agents is performed by the exchange of messages, defined as performatives ρ , ie. message types. Messages take a list of

terms as input $\phi^{(k)}$. Terms are defined as either a wildcard `(_)` , an agent name `(a)`, a role type `(r)`, a constant `(c)`, or a variable `(v)`. The send and receive actions contain two arguments $\phi_{(1)}$ and $\phi_{(2)}$. Agents can send a message to a specific agent (if $\phi_{(1)}$ contains an agent name), to any agent which is subscribed to a particular role (if $\phi_{(1)}$ is a wildcard and $\phi_{(2)}$ contains a role type), or simply send a message to any agent (if $\phi_{(1)}$ and $\phi_{(2)}$ are both wildcard types). Message passing between agents is assumed to be reliable, non blocking, buffered communication.

Control-flow in the protocol can be enforced in three ways. Firstly 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 parts of the protocol. If any action inside the loop body fails or the loops times out then the actions contained within the `timeout` body will be executed.

4.1 Protocol Execution

The MAP language is a specification designed to be directly executed by a group of agents. The typical process of executing a MAP interaction protocol is illustrated by figure 5.

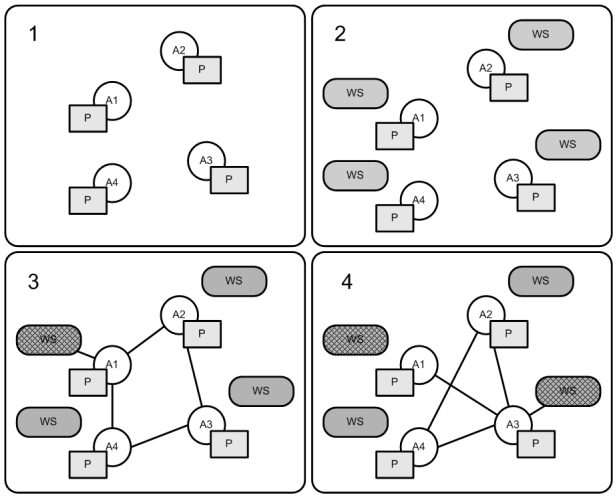

Fig. 5. MAP Protocol Execution

Once an engineer has designed a MAP interaction protocol, each agent taking part in the coordination must obtain a copy. This copy is stored locally to each

agent, illustrated by step 1 of figure 5. Agents are represented as a circle with (A) inside and the interaction protocol as a grey rectangle with (P) inside. 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.

Each agent taking part in the interaction must adopt a role, by adopting a role the agent must reference a reasoning web service which implements all the decision procedures required for that role type. This concept is illustrated by step 2 on figure 5: the reasoning web services are represented as a rounded rectangle containing (WS). This reasoning web service can be different for each agent. 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, calling the web services if and when required. Step 3 on figure 5 shows a pattern of interaction taking place, with the agent in the top left invoking its web service (hashed out on the diagram). A further pattern of interaction 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. Execution terminates when all the protocol steps have been enacted, or 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 Application of Framework to Scenario

This section further illustrates the MAP protocol language by applying it to the motivating scenario presented in section 3. Figure 6 is a MAP protocol definition of an agent attempting to classify some of the left over data from the subtracted image processing. For simplicity the protocol contains just one agent definition, the role of `classification`, however it interacts with agents who have adopted the `scientist`, `contractnet` and `observatory` roles. Firstly it is important to note that different types of term are represented by prefixing variable names with `$`, role names with `%` and agent names with `!`

The classification protocol, shown in figure 6 implements the classification agent process described in section 3.1 and proceeds as follows. A list of unclassified objects `($junk)` is received from a scientist agent (line 3). This list contains pointers to objects which cannot be classified by the automated algorithms discussed in the scenario. The agent then recursively traverses the list, attempting to classify the items locally. If at any time the agent cannot classify an object, calls to distributed observatories need to be made (line 14). This is achieved by making a request to an agent who has adopted the contractnet (line 21) role, supplying as parameters to the message: a list of suitable agents (`$potential_agents`, line 18) and a proposal (`$proposal`, line 19). The contractnet agent (not described in this example) executes the contract net pro-

```
%classification{
1   method() =
2    waitfor
3      (request($junk) <= agent($scientist, %scientist)
4       then invoke (localanalysis, $junk)
5       then invoke())
6    timeout (e)
7
8   method(localanalysis, $junk) =
9    (($head, $tail) = ExtractNext($junk)
10     then $result = StatTest($head)
11     then UpdateKnowledge($result)
12     then (QueryKnowledge($head)
13     then invoke(localanalysis, $tail)
14   or invoke(contractnetsend, $head, $tail))
15   or e)
16
17  method(contractnetsend, $unknown, $objects) =
18   $potential_agents = LookUp(%observatory, $unknown)
19   then $proposal = GenerateProposal($unknown)
20   then $cn = LookUp(%contractnet)
21   then request($potential_agents, $proposal) => agent($cn, %contractnet)
22   then waitfor
23     (response($open_proposals) <= agent($cn, %contractnet)
24      then ($accept, $reject) = Evaluate($open_proposals)
25      then invoke(contractaccept($accept, $objects, $unknown))
26         par invoke(contractreject($reject, $objects)))
27   timeout(e)
28
29  method(contractaccept, $accept, $objects, $unknown) =
30   ($observatory, $proposal) = ExtractProposal($accept)
31    then accept-proposal($proposal) => agent($observatory, %observatory)
32    then waitfor
33      (inform-result($opinion) <= agent($observatory, %observatory)
34       then $combined_opinion = GenerateOpinion($opinion)
35       then inform($combined_opinion) => (_, %scientist)
36       then invoke(localanalysis, $objects))
37      or (inform-failure() <= agent($observatory, %observatory)
38       then invoke(contractnetsend, $unknown, $objects))
39    timeout(e)
40
41  method(contractreject, $reject) =
42   ($head, $tail) = ExtractNext($reject)
43   ($observatory, $proposal) = ExtractProposal($head)
44    then reject-proposal($proposal) => agent($observatory, %observatory)
45    then invoke(contractreject, $tail)}.
```

Fig. 6. LSST Agent Protocol

tocol, contacting all observatory agents in the list `$potential_agents`. When finished the contractnet agent returns a list of observatory agents (line 23) who returned `propose` to the protocol. The list of open proposals is then evaluated locally (line 24), generating a list of rejected agents: `$reject` and a single suitable agent: `$accept`. An `accept-proposal` message is sent to the selected agent. If the observatory agent completes the tasks specified in the proposal an `inform-result` message is received (line 33). The data `$opinion` is used to generate a `combined_opinion` which is forwarded to the original scientist agent; informing a human scientist if anything unusual has occurred. A recursive call is then made, in an attempt to classify the remaining objects (line 36). However, if the observatory agent has been unsuccessful in completing its task, an `inform-failure` message is received (line 37). In this case, another attempt must be made to find suitable data from distributed observatories (line 38). In parallel to this task taking place, the agents who were unsuccessful in the proposal bid are rejected by the `reject-proposal` message (line 44).

The classification protocol is a straight forward implementation of the required functionality of the scenario, however there are some subtle issues in the protocol which require explanation. Role definitions can be divided up into a set of methods, allowing protocol code to be separated into smaller, manageable code chunks. Our protocol contains five method declarations. Protocols always begin execution with the default method, which is shown in this example from lines 1-6. Methods can be called by using the `invoke` operator, passing the necessary set of Terms as parameters to the method. For example, line 4 of the role definition shows an agent invoking the `localanalysis` method, using the list of unclassified objects, stored in the variable `$junk` as a parameter. An empty `invoke()` operation (line 5) will restart the default method when the protocol execution has terminated, effectively restarting the agent.

Agents connect to their internal reasoning layer by making invocations to a set of functions exposed as a reasoning web service. This set of functions implements a given role definition, so for our `classification` agent the web service contains the following functions: `ExtractNext, StatTest, UpdateKnowledge, QueryKnowledge, LookUp, GenerateProposal, Evaluate, ExtractProposal and GenerateOpinion`. Line 12 shows the `QueryKnowledge` function being invoked, using the `$head` variable as a parameter. As discussed briefly in section 4, control flow is enforced by the sequence (then), choice (or) or parallel (par) operators. The use of the sequence and choice operators is illustrated in the `localanalysis` method. The agent extracts the head: `$head` and tail: `$tail` of the list and attempts to classify the head of the list locally, by invoking the `StatTest` function (line 10). It then proceeds to query its local knowledge based on the updated information (line 12). If the `QueryKnowledge` function fails, the or branch of the protocol is executed, invoking the `contractnetsend` method, which begins to seek assistance from distributed observatory agents. However, if the `QueryKnowledge` function succeeds (the agent can classify the data) a recursive call `invoke(localanalysis, $tail)` is made, using the tail of the list as input. If the function `ExtractNext` fails, meaning that the list is now empty the

second or branch will be executed, in this case the empty action: `e`. The parallel operator is used in lines 25 and 26, in order to execute the `contractaccept` and `contractreject` methods.

The semantics of message passing corresponds 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(`$\phi_{(1)}$`,` $\phi_{(2)}$`)` is performed, where $\phi_{(1)}$ is matched against an agent name and $\phi_{(2)}$ to the agent role. For example, in line 3 of the protocol, the agent will receive the list of unclassified objects from any agent whose role is `%scientist`, and the name of this agent will be bound to the variable `$scientist`, for later reference. However in line 23, the classification agent will only receive the response from an agent of role observatory and in particular, the agent we sent the original request to, which is bound to the variable `$cn`. If the unification is successful, variables are bound based on the content of the message. For example, `$open_proposals` is stored locally upon receiving the message `response($open_proposals)`, shown in line 23. This unification is particulary useful when we do not know the exact name of the agent in question and simply want to receive a message from a particular role type.

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. Race conditions are avoided by wrapping all receive actions in `waitfor` loops. For example in line 3, the agent will continue to loop until a `request` message is received. If this loop was not present the agent may fail to receive the reply and the protocol would terminate prematurely. A further advantage of using non blocking communication is that we can check for a number of different messages. Inside the `waitfor` loop (lines 32-39) the agent waits for either an `inform-result` message, indicating the observatory agent has fulfilled the original proposal, or an `inform-failure` message. The flow of the protocol is very different, depending on which message is received. Timeouts, which have not been used in this protocol implementation, specify what to do if a timeout (specified by a time limit) is reached.

6 Conclusions and Further Work

This paper has demonstrated how scientific workflow, a problem from the Grid community can be elegantly modelled with the use of interaction protocols, a technique from the multiagent systems community. Our scenario demonstrates a number of runtime decisions which need to be taken, highlighting why a predefined static workflow cannot solve the service composition problem. Interaction protocols offer a flexible, adaptable solution to scientific workflow modelling.

In particular, the MAP language and framework allows complex multiagent interactions and web service invocations through the use of a relatively simple formalism. It offers a number of advantages over the coordination techniques used by existing projects focused at scientific workflow composition:

- **Reasoning Models:** The MAP approach allows the rules of interaction to be explicitly expressed, while allowing individual agents to subscribe to their own reasoning models. MAP 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 engineer of the agent to provide the set of methods which form this reasoning web service.
- **Inter-operability:** 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 within an open system, is a layer of software which can translate the protocol and a set of methods which make up the agents reasoning web service.
- **Layered Structure:** This model of interaction fills the gap between the low level transport issues of an agent and its high level rational processes. This layering removes some of the complications of designing large multiagent systems; ultimately helping in the design process.
- **Abstraction:** 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 to make decisions at runtime when the web service coordination is actually taking place. Decisions can be taken for example about: which services to call, what to do if a particular service is down, how to react if an expected message is not received etc. This approach offers more than 'just coordination', provided by most web service composition frameworks and languages.
- **Rapid Prototyping:** As the protocols provide an executable specification of the coordination, they serve as an excellent mechanism for rapidly prototyping a sequence of interaction. Protocols can be used to engineer a prototype system from a scenario, even if the services or interaction model, or even both are undefined at the design stage. Services can be stubbed.
- **Compatibility:** The coordination mechanism defined using the MAP 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.

This work forms part of an on going research and implementation process. Many enhancements to the language are in the process of being made that make

it more suited to e-Science computation. These enhancements include: support for large datasets through an extension of the type language; support for long-lived computation, e.g. by allowing break-points in the protocols; database integration for better handling of experiment data; and support for the composition of protocols into larger experiments at the scene level.

References

1. Business Process Execution Language for Web Services Specification, Version 1.1. Technical report, BEA Systems and IBM Corporation and Microsoft Corporation and SAP AG and Siebel Systems, July 2002.
2. Smart Grid Technologies Workshop. In *Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, Utrecht, The Netherlands, July 2005.
3. Agent-Based Grid Computing Workshop. In *6th IEEE International Symposium on Cluster Computing and the Grid*, Singapore, May 2006.
4. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 2004.
5. 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*, June 2004.
6. 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.
7. 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.
8. David Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition, Document Number tc00-1003 edition, January 1995.
9. Professor Dr. Huaglory and Dr. Rainer Unland, editors. *Multiagent and Grid Systems*. IOS Press.
10. B. Ludäscher, I. Altintas, and E. Jaeger-Frank M. Jones E. Lee J. Tao Y. Zhao C. Berkley, D. Higgins. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, Special Issue on Scientific Workflows, 2005.
11. 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.
12. Interaction Protocol Specifications. http://www.fipa.org/repository/ips.php3. Technical report, Foundation for Intelligent Physical Agents, 2002.
13. 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.
14. Robert Stevens, Robin McEntire, Carole Goble, Mark Greenwood, Jun Zhao, Anil Wipat, and Peter Li. ^{my}Grid and the Drug Discovery Process. *Drug Discovery Today: BIOSILICO*, 4(2):140–148, 2004.
15. Large Synoptic Survey Telescope. http://www.lsst.org.
16. 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.