Choreographing Web Services

Adam Barker, Christopher D. Walton, and David Robertson

Abstract—This paper introduces the Multiagent Protocols (MAP) Web service choreography language and demonstrates how service choreographies can be specified, verified, and enacted with a comparatively simple process language. MAP is a directly executable specification, services do not have to be preconfigured at design-time. Instead, a choreography, specified in MAP, can be sent dynamically to a group of distributed peers to execute at runtime. Furthermore, MAP is based on a formal foundation, this allows model checking of the choreography definition prior to live distribution and enactment. A motivating scenario, taken from the AstroGrid science use-cases, serves as the focal point for the paper and highlights the benefits of choreography, through data flow optimization and lack of centralized server. The MAP formal syntax and model checking environment are discussed in the context of the motivating scenario, along with MagentA, an implementation of MAP which provides a concrete, and open-source framework for the enactment of distributed choreographies. MAP is evaluated by demonstrating the languages conformance to the Service Interaction Patterns, a collection of 13 recurring workflow patterns.

Index Terms—Workflow, Web service choreography.

1 Introduction

Service-oriented architecture (SOA) is an architectural approach for the implementation and delivery of loosely coupled distributed services [34]. Although the concept of service-oriented architecture is not a new one, this approach has seen wide spread adoption through the Web services approach, which has a set of basic, core standards (XML, WSDL, SOAP, etc.) to facilitate service interoperability.

The core standards do not provide the rich behavioral detail which describes the role an individual service plays as part of a larger, more complex collaboration. This collaboration is often achieved through the use of workflow technologies. As defined by the Workflow Management Coalition [21], 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. Workflow can be described from the view of a single participant using *orchestration* or from a global perspective using *choreography*.

Service orchestration enables Web services to be composed together in predefined patterns, described using an *orchestration language* and executed on an *orchestration engine*. Orchestrations can span multiple applications and/or organizations and result in long-lived, transactional processes. Services themselves have no knowledge of their involvement in a higher level application, and therefore, need no alteration before enactment. Importantly, service orchestrations are described from the view of a *single participant* (which can be another Web service), and therefore, a central process always acts as a controller to the involved services. Orchestration languages explicitly describe the interactions between Web services by identifying messages, branching logic, and invocation sequences. The *Business Process Execution Language (BPEL)* [36] is an executable business process modeling language and the current de facto standard way of orchestrating Web services. BPEL has broad industrial support from companies such as IBM, Microsoft, and Oracle with concrete implementations.

Service choreography, on the other hand, is more collaborative in nature. A service choreography is a description of the peer-to-peer *externally observable interactions* that exist between services; therefore, choreography does not rely on a central coordinator. A choreography model describes multiparty collaboration and focuses on message exchange; each Web service involved in a choreography knows exactly when to execute its operations and with whom to interact. A choreography definition can be used at design-time to ensure interoperability between a set of peer services from a *global perspective*, meaning that all participating services are treated equally, in a peer-to-peer fashion.

There are two key approaches to modeling choreographies: *interaction models* and *interconnection models*. Interaction models use atomic interactions as the basic building blocks, and control and data flow are defined between them from a global perspective. Interconnection models, on the other hand, define control flow on a per participant basis. Corresponding send and receive activities are connected through a message flow, jointly representing interactions. For a detailed discussion of the differences, see in [18].

The *Web Services Choreography Description Language (WS-CDL)* [27] is a language to specify interaction models. More specifically, it is an XML-based language that can be used to describe the common and collaborative observable behavior of multiple services that need to interact, in order to

- *A. Barker is with the Department of Engineering Science, University of Oxford, Parks Road, Oxford OX1 3PJ, UK. E-mail: adam.barker@gmail.com.*
- *C.D. Walton is with Metaforic Ltd., Regent Court, 70 West Regent Street, Glasgow G2 2QZ, UK. E-mail: cwalton@gmail.com.*
- *D. Robertson is with Informatics, The University of Edinburgh, Informatics Forum, Crichton Street, Edinburgh EH8 9AB, UK. E-mail: dr@inf.ed.ac.uk.*

Manuscript received 5 Sept. 2008; revised 20 Nov. 2008; accepted 10 Mar. 2009; published online 17 Mar. 2009.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2008-09-0078.
Digital Object Identifier no. 10.1109/TSC.2009.8.

achieve a shared goal. WS-CDL is a W3C Candidate Recommendation.

To conclude, orchestration differs from choreography in that it describes a process flow between services from the perspective of one participant (centralized control), a *bottom-up* approach to design. Choreography, on the other hand, tracks a sequence of messages involving multiple parties (decentralized control, no central server), where no one party truly owns the conversation, a *top-down* approach to design, and an agreement between a set of services as to how a given collaboration should occur.

This paper introduces the Multiagent Protocols (MAP) Web service choreography language based on a formal foundation, π-calculus. MAP choreographies can be specified, verified through model checking, and enacted over a distributed peer-to-peer network. Cycling through the phases is supported, adding flexibility into the design process. Each of these phases are discussed in detail in the context of a motivating scenario taken from the AstroGrid science use-cases.

1.1 Motivating Scenario—Calculating Redshift

At this point, in order to put our motivation and problem statement into perspective, it is useful to consider a motivating scenario. The Redshift scenario is taken from the AstroGrid[1] science use-cases and involves retrieving and analyzing large-scale data from multiple distributed resources. This scenario will be addressed throughout the remainder of this paper.

Photometric Redshifts use broadband 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 scenario describes the process of querying a group of distributed databases containing astronomical images in different bandwidths, extracting objects of interest and calculating the relative Redshift of each object.

The scenario represents a workflow and begins with a scientist inputting the right ascension (RA) and declination (DEC) coordinates into the system, which define an area of sky. These coordinates are used as input to three remote astronomical databases; no single database has a complete view of the data required by the scientist, as each database stores only images of a certain waveband. At each of the three databases, the query is used to extract all images within the given coordinates which are returned to the scientist. The images are concatenated and sent to the SExtractor [9] tool for processing. SExtractor scans each image, in turn, and uses an algorithm to extract all objects of interest (positions of stars, galaxies, etc.) and produces a table for each of the wavebands containing all the data. A cross matching tool is then used to scan all the images and produce one table containing data about all the objects of interest in the sky in the five wavebands. This table is then used as input to the HyperZ[2] algorithm which computes the photometric Redshifts and appends it to each value of the table used as input. This final table consists of multiband files containing the requested position as well as a table containing for each source all the output parameters from SExtrator and HyperZ, including positions, magnitudes, stellar classification, and photometric Redshifts and confidence intervals; the final table is returned to the user.

1. http://www.astrogrid.org [16/12/2008].
2. http://webast.ast.obs-mip.fr/hyperz/ [16/12/2008].

1.2 The Case for Choreography

The majority of workflow research has focused on designing languages for implementing service orchestrations from the view of a single participant, where control and data flow pass through a centralized server. There are a plethora of orchestration frameworks which will automate these tasks, examples can be seen in the Business Process Modeling community through BPEL and life sciences community through Taverna [33].

Choreography, although an established concept is a less well researched and implemented architecture. In practice, the design processes and execution infrastructure for service choreography models are inherently more complex than orchestration; decentralized control brings a new set of challenges which are the result of message passing between distributed asynchronous, concurrent processes. However, although more complex, there are a number of arguments for adopting choreography.

Design argument. From a software design perspective, orchestration is suitable when the goal is to build individual services or to service-enable existing applications. However, during the early phases of service design, the emphasis lies not on building individual services but rather on how groups of services work together, by identifying collections of potential services and understanding and analyzing their interactions; at this early stage in the design process, engineers require a global view of how Web services interact with one another; choreography provides just these tools and is a description of multiparty collaboration.

Unbiased argument. Choreography is an unambiguous way of describing the relationships between services in a global peer-to-peer collaboration, without requiring orchestration at all. Each party takes an equal, predefined, and preagreed role in the choreography, this removes the scenario where businesses are interacting over a shared task but one organization has control over another by orchestrating their services.

Scalability argument. Centralized control through an orchestration engine is a valid solution for scenarios found in e-Commerce, where relatively small quantities of intermediate data (when output from one service invocation is directly, with no alteration, used as input to another) are moved between services in a workflow. However, centralized servers make less sense when dealing with data centric workflows (GBs/TBs), common to scientific applications. Passing large quantities of intermediate data through a centralized orchestration engine results in unnecessary data transfer, wasting bandwidth, overloading the engine, and decreasing the performance of a workflow. Furthermore, the orchestration engine becomes a single point of failure for the execution of a workflow.

Fig. 1a represents the motivating scenario where the involved services are orchestrated, both control and data flow pass through the centralized workflow engine. Each of the three astronomical databases are queried and return 100 Mb to the orchestration engine, these data (300 Mb) are

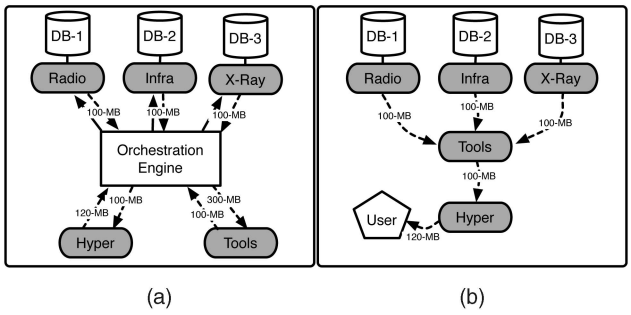


Fig. 1. Redshift scenario—(a) orchestration and (b) choreography.

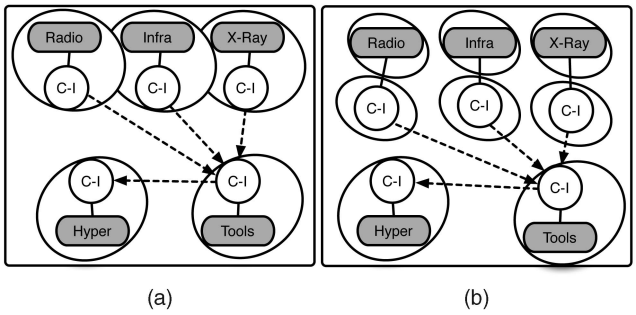


Fig. 2. Redshift scenario. (a) A configuration where all peers reside on the same server as the Web services they are invoking. (b) A mixed configuration, i.e., some peers are remote.

combined and sent to the SExtractor and cross matching tool (represented as Tools), the output of which (100 Mb) is returned to the orchestration engine and sent again as input to the HyperZ application. Finally, HyperZ computes the Redshift and tags the extra data to the input, returning 120 Mb to the orchestration engine. In order to enact the scenario a total of 920 Mb of data flow through the system.

By adopting a choreography model, the output of a service invocation can be passed directly (with some control flow between the services, omitted in Fig. 1) to where it is required as input to the next service, represented in Fig. 1b. As data do not have to be passed through a centralized engine, the choreography approach involves a total data transfer of 520 Mb, assuming that no data transformation (i.e., shims) takes place and the final output needs to be sent to the scientist. Using choreography, the total data transfer needed to enact the workflow is 400 Mb less when compared to orchestration.

1.3 Paper Contributions

This paper presents a Web services choreography language, MAP. MAP is an implementation of interconnection choreography models and is derived from process calculus, specifically, the π-calculus [32], see [37] for full details. It is important to note that we are not pitching MAP as a replacement to alternative specifications, such as WS-CDL. Rather this paper aims to demonstrate how service choreographies can be specified, verified, and enacted with a comparatively simple process language. The key contributions of this paper are summarized as follows:

Loosely coupled choreography interface. The Web Services Description Language (WSDL) specification provides a standardized interface description language to expose application code to a network, describing atomic, and low-level functions. WSDL was deliberately designed to be simple and lightweight and is a contributing factor to the success of Web services. However, WSDL does not provide the rich behavioral detail that describes the role a service plays as part of a larger, more complex collaboration. To enable Web services to collaborate autonomously, without centralized control, an extra layer of functionality, a *choreography interface* needs to be added to the stack.

In MAP, this extra functionality is achieved through the installation of a peer which sits in front of a service or group of services. Peers provide a choreography interface, exposed via WSDL, accessible, therefore like any other Web service. A peer is *decoupled* from the Web service definitions(s), by this we mean that individual Web services do not have to altered prior to enactment and services themselves require no knowledge that they are taking part in a more complex collaboration. A peer can be installed on the same server, domain, network, or externally from the Web services it is invoking. However, the optimal configuration in terms of data flow is that the communication between a peer and a Web service is local, i.e., not transferred over a Wide Area Network (WAN). Fig. 2 illustrates this concept using the Redshift scenario as an exemplar. In Fig. 2a, peers and services reside on the same server as one another; in Fig. 2b, the configuration is mixed, the Radio, Infra, and X-Ray peers are installed on separate servers but the Tools and Hyper peers sit on the same server.

Executable choreography language. MAP is an executable choreography language with multiparty (more than two partners) support. Peers act as blank canvases and do not have to be preconfigured with a particular choreography definition in advance at design-time. Instead, a choreography definition (defined in MAP) can be sent over the network to a group of peers dynamically, e.g., a choreography specifying an auction can be uploaded to a group of peers to enact, once terminated the same set of peers can be sent a completely different choreography specifying, for example, a business transaction. This provides a flexible solution and allows choreography definitions to be uploaded to a group of peers at runtime for execution. Even though MAP is a relatively simple language, this paper demonstrates how it still conforms to 12 out of the 13 Service Interaction Patterns, a set of recurring workflow patterns.

Verification through Model Checking. Building choreographies is a complex task, even for an experienced engineer. Asynchronous service choreography introduces nondeterminism into the system which causes a number of potential problems such as synchronization, fairness, and deadlocks. MAP is a directly executable specification based on a formal foundation, allowing us to provide a translation mechanism from MAP to PROMELA, the input of the SPIN model checker. Model checking allows verification of the choreography prior to enactment.

Open-source implementation. MAP is not merely a theoretical framework, a concrete implementation and framework for the enactment (not simulation) of distributed choreographies is provided, available as an open-source project. This implementation is based on Java, Web services, and XML technologies. Furthermore, it has been successfully applied to enact real choreographies on several e-Science projects.

$P \in$ Protocol	::=	$n\ (r\{M\})^{+}$	(Choreography)
$M \in$ Method	::=	`method`$(\phi^{(k)})$ = op	(Method)
$op \in$ Operation	::=	α	(Action)
	|	op_1 `then` op_2	(Sequence)
	|	op_1 `or else` op_2	(Choice)
	|	op_1 `par` op_2	(Parallel Composition)
	|	`waitfor[:imax]` op_1 `timeout[:tmax]` op_2	(Iteration)
	|	`call`$(\phi^{(k)})$	(Recursion)
	|	(op)	(Precedence)
$\alpha \in$ Action	::=	ϵ	(No Action)
	|	$\phi^{(k)}$ = `service`(ws^{+}, $\phi^{(l)}$) `fault` $\phi^{(m)}$	(Service Invocation)
	|	$p(\phi^{(k)})$ => `peer`$(\phi^{(1)}, \phi^{(2)})$	(Send)
	|	$p(\phi^{(k)})$ <= `peer`$(\phi^{(1)}, \phi^{(2)})$	(Receive)
$\phi \in$ Term	::=	$\_ \mid p \mid r \mid c : \tau \mid v : \tau$	(Terms)
$\tau \in$ Type	::=	$ptype \mid rtype \mid rpctype$	(Types)
ws	::=	`def`(config $^{(k)}$)	(Web Service Definition)
config	::=	⟨name, value⟩	(Configuration Pair)

Fig. 3. MAP syntax.

1.4 Paper Overview

The remainder of this paper is structured as follows. Section 2 discusses the MAP syntax in detail and describes how a choreography is built from a set of interacting peers. Section 3 introduces the protocol methodology which describes the tasks involved with taking a software specification and implementing a choreography in MAP. We illustrate the methodology through example, by providing a MAP implementation of the motivating Redshift scenario. Model Checking techniques are introduced which allow validation of a choreography before deployment. Section 4 describes MagentA, a Java/XML/Web services framework for enacting choreographies specified in MAP. Section 4.1 discusses how MagentA enacts a choreography across a group of distributed peers, i.e., peer location, peer routing, etc. Section 5 benchmarks MAP by discussing MAPs compliance to the set of Service Interaction Patterns, a type of Design Pattern for the workflow community. Section 6 discusses all choreography related work. Finally, Section 7 concludes the paper and discusses in context how MAP is differentiated from existing choreography solutions.

2 Multiagent Protocols (MAP)

The syntax of MAP is shown in Fig. 3. We note that MAP is only intended to express Web service choreographies and is not intended to be a general purpose programming language. Where examples are given, variables are represented by `$`, constants by `!`, and role types by `%`. For readability throughout the examples, constants are also used to abstract the details from Web service definitions, e.g., `!S1` contains the relevant PortType, OperationName, etc. Web Service definitions are distinguishable from other constants as they are always written in CAPITALS. Protocol code is numbered and referenced throughout the text. To highlight message passing, lines containing a send or receive action are marked in **bold** font. Protocol code in the main body of text is marked up using the `true type` font. For readability, no type definitions have been included in the MAP protocols.

2.1 Choreography: A Protocol, Roles, and Peers

A choreography in MAP is specified through a *protocol*, which is uniquely named `n`. A protocol is broken down into a number of distinct *roles* which *peers* adopt. A protocol can be thought of as a bounded space in which a group peers interact on a single task. MAP protocols add a measure of security, in that peers which are not relevant to the choreography are excluded from taking part in the interaction. We assume that a MAP protocol places a barrier on the peers such that enactment cannot begin until all the peers have been instantiated.

The concept of a role is central to our definition. In MAP, each peer is identified by both a unique name `p` and a role `r`. Peers are uniquely named, but must be assigned a role which is specified in the protocol definition. The role of a peer is fixed until the choreography has terminated and determines which parts of the protocol the peer will follow. Peers can share the same role, which defines them as having the same capabilities, i.e., the same interface. Roles are useful for grouping similar peers together, for example, we may wish to interact with a large number of peers, all with the same interface. Roles also allow us to specify multicast communication in MAP, for example, we can broadcast messages to all peers of a specific role.

2.2 Roles Defined through Methods

A role's behavior is defined by a set of Methods `{M}`, which can optionally take a list of Terms as arguments $\phi^{(k)}$. Terms are the objects of manipulation in MAP and are defined as either a wildcard `_`, a peer type (unique peer name) `(p)`, a role type (unique role name) `(r)`, a constant `(c)`, or a variable `(v)`. Variables are bound to terms by unification which occurs in the invocation of Web services, the receipt of messages, or through recursive `method` invocations, discussed later in the Section. Constants and variables are assigned types τ to ensure that they are treated consistently.

Types τ are defined through a peer type (`ptype`), which is a unique peer name and equivalent to `p`, or a role type (`rtype`), a unique role name and equivalent to `r`. Peer types and role types are useful when storing the unique name of peer or role locally for reference later in the protocol enactment. The final type is the `rpctype`, a type which conforms to the standard set of JAX-RPC support types.[3] This allows peers to store, for example, an Array of Strings.

Methods are constructed from an Operation Set `op`, which enforce control flow and a set of actions α, which allow the peers to send and receive messages to one another and invoke third part Web services. Actions can fail (e.g., failure to receive an incoming message, failure to invoke a Web service, etc.), failure of actions causes backtracking of the protocol.

2.3 Web Service Invocations

The action set first consists of the `service` action which allows peers to synchronously call Web services directly from within the protocol code. A Web service `ws` is specified using a list of configuration pairs $def(config^{(k)})$ which are generic $\langle \text{name}, \text{value} \rangle$ tuples used to define the details of service invocations, for example, the WSDL, PortType, etc. Multiple `ws` definitions can be used as the first parameter to a `service`. The first `ws` definition is used as the default service to invoke, the remainder act as backup services, called in the event that a fault arises with the first, although more dynamic solutions can be encoded into the protocol, i.e., a registry lookup returning a functionally equivalent service. 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 $\phi^{(m)}$.

2.4 Message Passing

Communication between peers is performed by message passing, defined as performatives ρ, i.e., message types. By *performatives*, we refer to a common format for the interchange of messages between agents/peers, for example, the Foundation for Intelligent Physical Agents (FIPA) Agent Communication Language (ACL) [1]. Messages take a list of terms as input $\phi^{(k)}$. The send and receive actions contain two arguments $\phi^{(1)}$ and $\phi^{(2)}$ and can be configured in a number of ways:

Specific peer, specific role. If the first parameter contains a peer type and the second parameter contains a role type. For example, `request($var1)` $\Rightarrow$ `peer($p1, %role1)` would send the message of performative type `request` containing `$var1` to the peer `$p1` who has adopted the role `%role1`. This feature is useful for sending messages to specific peers (who are known in advance or looked up at runtime), e.g., to maintain a long-running, consistent dialogue.

Specific peer, any role. If the first parameter contains a peer type and the second parameter contains a wild card. For example, `request($var1)` $\Rightarrow$ `peer($p1, _)` sends a message of performative type `request` directly to the peer represented by the variable `$p1`.

Any peer, specific role. As there is the possibility that many peers have adopted the same role, a useful feature is the ability to send and receive messages from any peer who has subscribed to a particular role. This is achieved if the first parameter contains a wildcard and the second parameter contains a role type. For example, `request($var1)` $\Rightarrow$ `peer(_, %role1)` would send the message of performative type `request` to any peer who has adopted the role `%role1`. This allows an engineer to specify multicast communication in MAP.

Any peer, any role. If both parameters are wild cards: `_`, for example, `request($var1)` $\Rightarrow$ `peer(_, _)`.

The semantics of message passing correspond to nonblocking, reliable, and buffered communication. Sending a message succeeds immediately if a peer 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 `peer`$(\phi^{(1)}, \phi^{(2)})$ is performed, where $\phi^{(1)}$ is matched against a peer type and $\phi^{(2)}$ to the role. If the unification is successful, variables are bound based on the content of the message $\phi^{(k)}$ and stored locally to the peer, for further use in the protocol. Sending will fail if no peer 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., nonblocking) and do not delay a peer.

2.5 Control Flow

Control flow in a protocol can be enforced in three ways. First, 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 else` op_2 handles failure in the protocol and evaluates op_2 only if op_1 contained an action that failed. MAP includes backtracking, such that the execution will backtrack to the nearest or operator when a failure occurs. The parallel operator op_1 `par` op_2 executes op_1 and op_2 in parallel. The parameter semantics of MAP are call-by-name. The operations in MAP are normally evaluated in strict left-to-right order. However, the precedence of the operations can be explicitly defined using parenthesis `(op)`.

A `waitfor` loop allows repetition of parts of the protocol and will be repeatedly executed upon failure. The loop will terminate when the body succeeds. A `waitfor` loop can also include a `timeout` condition which is triggered after a certain interval, `tmax` has elapsed. `tmax` is specified in seconds and armed once the loop begins. If a loop times out, then the actions contained within the `timeout` body will be executed, this includes the empty

3. http://ws.apache.org/xmlrpc/types.html [16/12/2008].

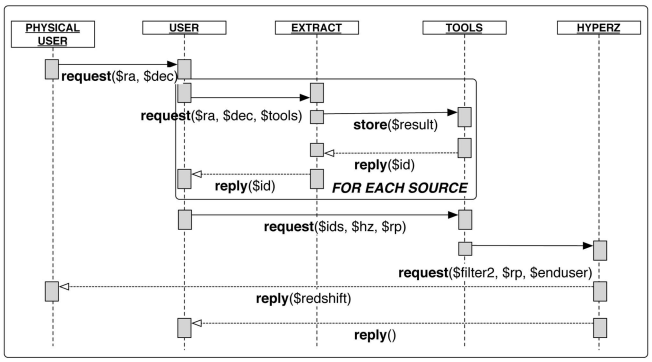


Fig. 4. Redshift scenario—UML sequence diagram.

action ϵ. Alternatively, the optional `imax` value can be set with an integer number, indicating how many times the loop will iterate; `imax` is not set by default. Repetition has been included in the language definition first because receive actions are commonly wrapped with a `waitfor` loop in order to synchronize the message passing between peers. Second, timeouts allow compensation actions to be defined as they are only executed if the loop times out, for example, what protocol code to execute if a message (which was expected) did not arrive. A recursive `method` invocation is achieved through the `call` operator. Calls are performed by pattern matching (Prolog-style), and the first argument in the pattern is the method name.

3 MAP Example—Redshift Scenario

The *protocol methodology* describes the task of writing a protocol to coordinate multiple, concurrent peers. 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 detailed in the following Sections and implements the motivating Redshift scenario detailed in Section 1.1.

3.1 Define Role Set

Role types specify a pattern of computational behavior which a peer can adopt. The first task an engineer must perform is to break the initial specification into a number of role types which together define a protocol and implement a choreography. This could be a single role, or multiple roles which are expected to interact as part of a more complex choreography.

Returning to our motivating scenario discussed in Section 1.1, the following roles are defined: The `user` role coordinates the activities of the other peers, receives input, and sends output to the physical user (i.e., a scientist, etc.) grounded in this scenario; the corresponding MAP code is shown in Fig. 5. The `extract` role retrieves and formats data from a number of distributed astronomical databases, represented in MAP in Fig. 6. Finally, the `tools` and `hyperz` roles manage groups of computational services, such as the cross matching tool and Redshift analysis, represented in MAP in Fig. 7.

```
1  %user{
2  method main() =
3   waitfor
4    (request($ra, $dec) <= peer($rp,_)
5    then $peers = service(def(!registry),!r, !i, !x)
6    then $tools = service(def(!registry), !tools)
7    then call uloop($ra, $dec, $peers, $rp, $tools)
8    then call main())
9   timeout(e)
10
11 method uloop($ra, $dec, $peers, $rp, $tools) =
12  (($h, $t) = Next($peers)
13                          fault emptylist
14  then request($ra,$dec,$tools) => peer($h,%extract)
15  then call uloop($ra, $dec, $t, $rp, $tools))
16    or else call uwait($rp, $tools)
17
18 method uwait($rp, $tools) =
19  waitfor
20   (reply($id) <= peer(_,%extract)
21   then service(def(!concatID), $id)
22   then call uwait($rp, $tools))
23  timeout(call uterminate($rp, $tools))
24
25 method uterminate($rp, $tools) =
26  $hz = service(def(!registry), !hyperz)
27  then $ids = service(def(!retrieveID))
28  then request($ids,$hz,$rp) => peer($tools,%tools)
29   then waitfor
30    (reply() <= ($hz, %hyperz))
31   timeout(e)}
```

Fig. 5. `user` peer MAP specification.

3.2 Define Role Interactions

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 roles within the protocol. The sending and receiving actions can (if necessary) be sugared with control flow (then, or, par, etc.). Fig. 4 is an UML Sequence diagram representing the message passing between the peer roles, performatives and parameters are presented in each of the messages.

3.3 Define Role Behavior

Once a set of roles have been defined and the interactions between them is coherent, an engineer must fill in the role type definitions. Each role is broken down into a group of `methods`, making use of the remainder of the action set and control flow operators. Engineers must consider how roles connect to any external services, taking into account input and output parameters along with any fault information, how to enforce reliability, etc. A MAP implementation of the peer roles are represented in Figs. 5, 6, and 7. These protocols implement the role interactions illustrated in Fig. 4.

With reference to Fig. 4 and the corresponding MAP syntax, the following pattern of interaction takes place. The `user` role enters the `main method` and immediately enters a `waitfor` loop, it is waiting for a message of performative type `request` (line 4 in Fig. 5) containing two parameters from any peer subscribing to any role, this is indicated by the wildcard in the second parameter. Once a message matching, this template is received the unique name of the peer `$rp` and the coordinates `$ra` and `$dec` are bound to

```
1  %extract{
2  method main() =
3   waitfor
4    (request($ra, $dec, $tools) <= peer($rp,_)
5    then call eloop($ra, $dec, $rp, $tools)
6    then call main())
7   timeout(e)
8
9  method eloop($ra, $dec, $rp, $tools) =
10  ($next = service(def(!REGISTRY))
11                                fault emptylist
12  then $result = service(def($next), $ra, $dec)
13  then store($result) => peer($tools,%tools)
14  then call eloop($ra, $dec, $rp, $tools))
15    or else call ewait($rp, $tools)
16
17 method ewait($rp, $tools) =
18  waitfor
19   (reply($id) <= peer($tools, %tools)
20   then reply($id) => peer($rp, _)
21   then call ewait($rp, $tools))
22  timeout(call main())}
```

Fig. 6. `extract` peer MAP specification.

local variables. Using the newly received coordinates (corresponding to an area of sky) a call to an external Web service is made in order to obtain a list of astronomical databases (line 5 of Fig. 5). The definition of the registry Web service is used as the first parameter to the `service` operator and the remaining three parameters form the corresponding input, radio `!r`, infrared `!i`, and X-ray `!x`, i.e., we are interested in extracting data from all sources. The results of the service invocation are bound to the local variable `$peers` which represents a list of peers managing databases containing radio, infrared, or x-ray astronomical images. A second service call is made in order to locate a suitable peer who has subscribed to the `tools` role. Once completed, the `call` operator is called on the `method uloop` (line 7 of Fig. 5) passing in as parameters the RA and DEC coordinates `$ra, $dec`, list of available peers `$peers`, the unique name of the requesting peer `$rp` and the unique name of the tools peer `$tools`. Control passes to the `uloop method` which retrieves the head and tail of the peers list through a built in language function call `Next`. A message of performative type `request` (line 14 of Fig. 5) is sent to the peer represented by the newly retrieved head of the list (indicated by the first parameter) which is subscribed to the role `extract` (indicated by the second parameter). In this case, the peer indicated by `$h` is determined by a runtime lookup and has not been statically coded into the protocol. The `uloop method` will recursively iterate until the `emptylist` fault is thrown; at this point, the `or else` branch will be executed and the `uwait method` will be called (line 16 of Fig. 5).

The `extract` peer enters the `main method` and immediately waits for a message of performative type `request`; notice that the receive on the `extract` peer (line 4 of Fig. 6) matches the send on the `user` peer (line 14 of Fig. 5). Once the peer receives the message, the variables are bound locally and an invocation to the `eloop method` is made. A call to a registry is made to obtain a physical resource (i.e., database) the peer is managing. Once a source is located, a service invocation is made (line 12 of Fig. 6) passing in the RA and DEC coordinates as parameters, the output (in this case a set of images) is bound to the local variable: `$result` and sent directly to the `tools` peer for processing (line 13 of Fig. 6). By sending the results directly to the `tools` peer, the intermediate results do not have to be sent via the `user`, effectively avoiding a network hop. The `eloop method` iterates until the service invocation throws an `emptylist` fault indicating that there are no more resources available for query. The `tools` peer is waiting for two types of message, either `store` (line 4 of Fig. 7) or `request` (line 7 of Fig. 7), divided with an `or else` control flow operator. This feature allows a peer to execute sections of a protocol depending on which type of message it receives and which role the sender is subscribed to.

```
1  %tools{
2  method main() =
3   waitfor
4    ((store($data) <= peer($rp,_)
5    then call store($data, $rp)
6    then call main())
7   or else request($ids,$hz,$enduser) <= peer($rp,_)
8    then call process($ids, $hz, $enduser, $rp)
9    then call main())
10  timeout(e)
11
12 method store($data, $rp) =
13  $id = service(def(!GENERATEID))
14  then service(def(!STORE), $data, $id)
15  then reply($id) => peer($rp,_)
16
17 method process($ids, $hz, $enduser, $rp) =
18  $raw_data = service(def(!RETRIEVE), $ids)
19  then $f1 = service(def(!SEXTRACTOR), $raw_data)
20  then $f2 = service(def(!MATCHER), $f1)
21  then request($f2, $rp, $enduser) => peer($hz,_)
22
23 %hyperz{
24 method main() =
25  waitfor
26   (request($data, $rp, $enduser) <= peer(_,_)
27   then $Redshift = service(def(!HYPERZ), $data)
28   then reply($Redshift) => peer($enduser,_)
29   then reply() => peer($rp,_)
30   then call main())
31  timeout(e)}
```

Fig. 7. `tools` and `hyperz` peer MAP specifications.

Messages of performative type `store` are sent from the `extract` peer to the `tools` peer in order to house the data before processing begins, notice that again the receiving signature (line 4 of Fig. 7) matches the sending signature (line 13 of Fig. 6). Once a message of type `store` is received, the `store method` is called passing in the actual data and the unique name of the recipient peer as input. Control passes to the `store method`, an ID is generated and the intermediate data are stored at the peer through two Web service invocations. Once complete, the corresponding unique identifier is passed back to the `extract` peer that sent the request, indicated by the use of the `$rp` variable in the first parameter (line 15 of Fig. 7). Control then passes back to the `main method` which restarts the peer by recursively invoking the `main method` (line 6 of Fig. 7).

At this point, the `extract` peer is waiting for responses in the `ewait method`. Once it receives a message of performative type `reply` from a peer matching the unique name represented by the `tools` variable, it is forwarded back to the `user` peer (lines 19 and 20 of Fig. 6), represented by `$rp`. In order to wait for all messages, a recursive `call` is made to the `ewait method` (line 21 of Fig. 6) until the `waitfor` loop finally times out, once this happens, the code in the `timeout` clause is called and the peer is restarted (line 22 of Fig. 6).

The `user` peer is waiting for responses from the `extract` peer (line 20 of Fig. 5), once received the IDs are concatenated into a list, through an external service invocation and a recursive `call` is made to the `uwait method`. This `method` continues to execute until the `waitfor` loop eventually times out and the `uterminate method` is called (line 23 of Fig. 5).

Control then passes to the `uterminate method`. Through a registry call, a suitable `hyperz` peer is located to execute the Redshift calculation. The IDs (relating to the data stored at the `tools` peer) are retrieved and sent (line 28 of Fig. 5) along with the unique name of the `hyperz` peer and the end user. Once received by the `tools` peer (line 7 of Fig. 7), the second part of the `or else` branch is executed and the `process method` is called. The data are retrieved and passed through the SExtractor (in order to extract all objects of interest) and the cross matching tool (to combine the results) before being sent to the `hyperz` peer indicated by the `$hz` variable. The final Redshift calculation is performed on the combined data (line 27 of Fig. 7) and the results are sent directly to the user who requested the processing (line 28 of Fig. 7), indicated by the `$enduser` variable, a confirmation of protocol completion is also sent to the `user` peer. Execution of the protocol terminates.

3.4 Protocol Verification

An important consideration when defining service choreography is the issue of *correctness*. That is, if we are dynamically connecting together a collection of services to perform a task, we want to be reasonably sure that the services will interact correctly to accomplish this task. This is particularly true for long-lived computations that cannot trivially be repeated. We do not want the choreography to fail unexpectedly after a significant computational effort has been expended. Therefore, in this Section, we outline a technique that can be used to determine the correctness of MAP service choreography.

A key feature of protocols in MAP is that they are directly executable. Each participant in the interaction executes their role in the protocol, causing the interactions to happen at the right time and in the correct sequence. However, it is important to appreciate that the protocols are executed *concurrently*, by many participants at the same time. One participant may interact simultaneously with many others, and these interactions may be interleaved in complex ways. This concurrent behavior causes complications when we attempt to determine protocol correctness.

The key difficulty lies in the asynchronous nature of service choreography. Asynchrony introduces *nondeterminism* into the system which gives rise to a large number of potential problems, such as synchronization, fairness, and deadlocks. It is difficult, even for an experienced protocol designer, to obtain a good intuition for the behavior of a concurrent protocol, primarily due to the large number of possible interleavings that can occur. Debugging and simulation techniques cannot readily explore all of the possible behaviors of such systems, and therefore, significant problems can remain undiscovered.

To show the correctness of MAP service choreography, we turn to software verification techniques. The execution behavior of a complex service choreography is very similar to that of a concurrent (e.g., multithreaded) software application. This leads us naturally to model checking techniques [12], which are one of the main ways that the execution of concurrent software (and hardware) systems is verified. Model checking works simply by enumerating the state space of the system and checking its behavior along all possible execution paths. Given sufficient resources, the model checking process will always terminate with a yes/no answer. Model checking has been applied with considerable success to the verification of concurrent hardware systems, and it is increasingly being used as a tool for verifying concurrent software systems, including multiagent systems [8], [10], [38].

To perform model checking on MAP, we require an encoding of the service choreography into a form suitable for model checking. In [37], we previously defined a translation from MAP into PROMELA, which is the input language of the SPIN model checker [22]. A similar technique has been defined for the AgentSpeak language [10]. It is also helpful to sketch the translation of the semantics of MAP into modal temporal logic, as this underlies the model-checking process. We require only one modal construct: the term $\diamond\varphi$ denotes that the expression φ is true at some future time. Fig. 8 illustrates the translations into this form for the operations of MAP. The square brackets indicate that the translation should be applied recursively. The global environment Δ stores mappings from method arguments to operations ($\phi^{(k)} \mapsto op$). When a method is called, a pattern-matching operation is performed over the arguments in the environment $\Delta(\phi^{(k)})$, and the matching operation is evaluated. If no match can be found, the model checking process will fail.

A key feature of the MAP encoding process is the treatment of the actions. We make the observation that the purpose of each action is to impose a true/false decision on a protocol, and the purpose of the model checking process is to detect errors in the protocol and not in the services. Thus, based on these observations, we can replace each action with a pair of states, one of which signifies that the outcome is true, and the other false. The exhaustive nature of the model checking process means that all possible behaviors of the protocol are explored. In other words, the model checker explores all consequences for the protocol where the action was true, and all consequences where the action was false. Thus, we do not need to invoke the actual services, or perform any message-passing, during the model checking process.

The encoding which we have outlined here can be performed automatically by a computer as it does not require any specific knowledge about the protocol. This

$$
\begin{array}{rcll}
n(r\{M\})^{+} & \leadsto & [\![M^1]\!] \wedge \cdots \wedge [\![M^k]\!] & \text{(Choreography)} \\
\texttt{method}(\phi^{(k)}) = op & \leadsto & [\![op]\!] \quad \Delta \cup \{\phi^{(k)} \mapsto op\} & \text{(Method)} \\
\alpha & \leadsto & \bot \mid \top & \text{(Action)} \\
op_1 \texttt{ then } op_2 & \leadsto & [\![op_1]\!] \wedge \Diamond [\![op_2]\!] & \text{(Sequence)} \\
op_1 \texttt{ or else } op_2 & \leadsto & [\![op_1 \wedge \neg op_2]\!] \vee [\![\neg op_1 \wedge op_2]\!] & \text{(Choice)} \\
\texttt{waitfor } op_1 \texttt{ timeout } op_2 & \leadsto & \Diamond([\![op_1 \wedge \neg op_2]\!] \vee [\![\neg op_1 \wedge op_2]\!]) & \text{(Iteration)} \\
\texttt{call}(\phi^{(k)}) & \leadsto & [\![\Delta(\phi^{(k)})]\!] & \text{(Recursion)}
\end{array}
$$

Fig. 8 MAP formal semantics.

makes the technique suitable for use by nonexperts who do not need to understand the model checking process. However, we note that this approach places restrictions on the kinds of properties of the protocols that we can check. In particular, we cannot automatically verify properties which are specific to the domain of the protocol.

We have principally focused on checking the *termination* of MAP service choreography with model checking. This is an important consideration in the design of protocols, as we do not (normally) want to define protocols that cannot conclude. Nontermination can occur as a result of many different issues such as deadlocks, live-locks, infinite recursion, and message synchronization errors. Furthermore, we may also wish to ensure that protocols do not terminate due to failure within the protocol.

The termination condition is the most straightforward to verify by model checking. Given that progress is a requirement in almost every concurrent system, the SPIN model checker automatically verifies this property by default. The termination condition states that every process eventually reaches a valid end state. System properties are expressed in Linear Temporal Logic (LTL) inside the SPIN model checker. The termination can be expressed as the following LTL formula, where `end1` is the end state for the first process, and `end2` is the end state for the second process, etc., $\Box(\Diamond(\texttt{end1} \wedge \texttt{end2} \wedge \texttt{end3} \wedge \cdots))$.

One of the main pragmatic issues associated with model checking is typically producing a state space that is sufficiently small to be checking with the available resources. Hence, it is often necessary to use abstraction techniques, such as we have done for the actions, and to make simplifying assumptions. Other researchers have also considered this problem. For example, Bordini et al. [11] propose a program-slicing technique to improve the efficiency of the model checking process. Nonetheless, in our experience, MAP specifications tend to be compact even for a complex choreography, and we have not encountered any significant difficulties checking very complex protocols, such as the Redshift scenario described previously. As a result, we claim that model checking is a very useful and powerful automated technique for verifying the correctness of service choreography. For a detailed description of the model checking techniques discussed in this paper, refer to a complimentary paper [37].

4 MAP Implementation

The MAP specification has been implemented as an open-source Web service choreography framework, MagentA[4] using a combination of Java (J2SE v1.4.2 SDK), Web services (Java Web Services Developer Pack 1.5), and XML technologies. A peer is a lightweight, noninvasive piece of middleware that serves as a proxy to a Web service or group of Web services. For optimized data flow, a peer should be deployed on the same server, network, or domain as the Web service(s) it is invoking, so that communication between a peer and a Web service is conducted via a Local Area Network not a Wide Area Network. The MAP language has been represented using an XML schema, providing a straightforward conversion from the formal syntax to computer interpretable form.

The interface to a peer is exposed via WSDL and is therefore accessible like any standard Web service. This flexibility allows a gradual change of infrastructures, where one, could for example, concentrate first on improving data transfers between services that handle large amounts data.

4.1 MAP Enactment Process

Once a MAP protocol is defined through the protocol methodology described in Section 3, it must be disseminated to a group of peers for enactment. Engineers write MAP protocols in the XML encoded syntax, this is automatically translated to the MAP syntax. As discussed in Section 1.3, MAP protocols are directly executable choreography specifications. Peers are, in effect, blank canvases and can execute any MAP protocol, i.e., choreography specifications do not have to be hard-coded into a peer at design-time. This allows protocols to be uploaded and executed dynamically. Fig. 9 illustrates the steps involved with protocol enactment. With reference to Fig. 9, the following process takes place.

Locate peers. Prior to the dissemination of a protocol, peers are located at runtime through a registry lookup service. The physical network location (IP address) of each peer willing to fulfill a role defined in the protocol is spliced into the MAP protocol definition before it is disseminated to the participating peers. All roles defined within the protocol must be filled, e.g., for our Redshift scenario, the `user`, `extract`, `tools`, and `hyperz` roles. The physical network locations of peers facilitate message routing, allowing peers

4. http://homepages.inf.ed.ac.uk/cdw/ [16/12/2008].

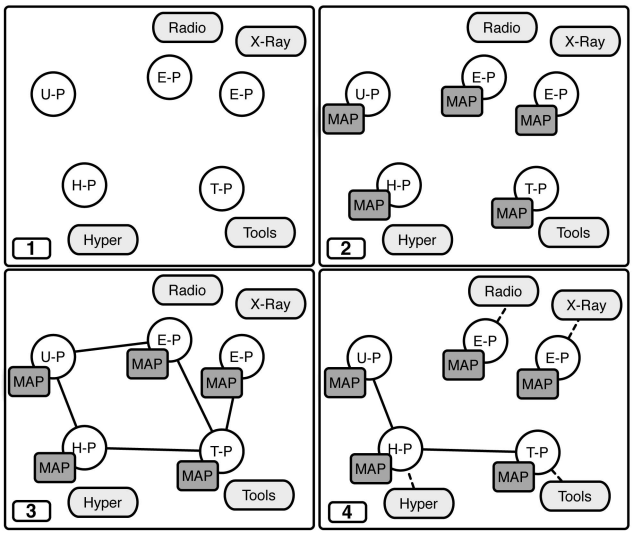


Fig. 9. The MAP execution process. Peers are represented as a circle, corresponding to the User (U-P), Extract (E-P), Tools (T-P), and HyperZ (H-P) peers. A protocol (described using MAP) is represented as a rectangle and Web services as rounded circles. Step 1 illustrates peers sitting in front of standard Web services. In Step 2, the MAP protocol is disseminated to each peer willing to fulfill a role in the protocol. Step 3 illustrates an example pattern of interaction between peers and Step 4 illustrates a further pattern of interaction involving external Web service invocations.

to send and receive messages to one another. More details of this process are described in a complimentary paper [28].

Receive protocol. Each peer willing to fulfill a role is sent the MAP protocol definition (encoded in XML) across the network, along with the role name it is required to adopt within the protocol. This is illustrated by steps 1 and 2 of Fig. 9.

Unpack protocol. Once a peer receives a MAP protocol, the parsing component will unmarhsall and validate it against the MAP XML Schema. Any exceptions through a malformed protocol are thrown to the exception handler, initialization is terminated, and exceptions are reported to the user.

Build execution model. If the validation is successful, the XML parser (implemented through JAX-B) converts the role 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 role definition.

Initiate interaction. Peers form a peer-to-peer system. As each peer has a local copy of the protocol, no centralized control is required. Once all peers have obtained a copy of the protocol and have been initialized, enactment of the choreography can begin.

Peers follow the protocol as a script, invoking actions (from the action set), Web services, and sending/receiving messages to one another. The MAP protocol defines which peer role initiates the choreography, i.e., an engineer specified this when constructing the protocol. Steps 3 and 4 of Fig. 9 illustrate an example pattern of interaction between the peers involved in the motivating Redshift scenario. Service invocations are handled by the JAX-RPC interface. This handler is generic in that it can call any method once it has obtained the WSDL definition. Details of which peers are enacting the MAP protocol were spliced into the MAP protocol definition before it was disseminated. Each peer has a local copy of the protocol, contained within that definition are the concrete details of where collaborating peers are located, i.e., an IP address. When a peer wants to send a message to another peer, these concrete network locations are addressed. Messages sent from one peer to another are encoded in XML and sent via SOAP. 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 MAP Validation

Service Interaction Patterns [7] (a subset of workflow patterns research) are a collection of 13 recurring patterns derived from insights into business-to-business transaction processing, use-cases gathered by standardization committees, generic scenarios identified in industry standards, and case studies reported in the literature. The collection of Service Interaction Patterns facilitate the assessment of emerging Web services standards by providing a common basis on which workflow languages can be compared. MAP is a comparatively simple process language and is therefore relatively sparse in features. MAP does not have dedicated constructs for dealing with these common interactions, i.e., no multicast support, no atomic transaction support. However, MAP can implement the majority of 13 Service Interaction Patterns by combining actions and operations together in novel ways. The solutions presented are not the only possible implementations.

The three simple patterns **Send**, **Receive,** and **Send/recieve** are directly supported in MAP through `send (=>)` and `receive (<=) actions`. MAP allows that a receiver of a message is bound at design-time or at runtime through a variable assignment.

In the case of the **Racing incoming messages** pattern, a party expects to receive one among a set of messages. MAP solution: A number of `receive (<=)` actions, each with their own message signature, i.e., performative type and message contents are separated by an `or else` operator, e.g., `type1($c1) <= peer(_,_) or else type2($c1, $c2) <= peer(_,_)`. Receive actions are usually wrapped with a `waitfor` loop with an optional `timeout` set. Messages from arbitrary senders can be received as long as the message signature matches that in the protocol, any peer type and role type constraints are met. This pattern was implemented in our the `tools` peer, lines 1-21 of Fig. 7.

One-to-many send. A party sends messages to several parties, the messages all have the same type, although their contents may be different. MAP solution: This pattern is supported by invoking the `call` operator recursively on a list of participants. A `method` is defined which recursively sends a message to a list of participants. The participant list is broken up into `$head` and `$tail`, a message is sent by splicing in the head of the list in the `send` action, e.g., `=> peer($head, _)`. The `call` operator invokes the `method` using the `$tail` of the list as input. This pattern was implemented in our motivating Redshift scenario, in particular, the `user` peer, lines 11-16 of Fig. 5. Alternative

MAP solution: define a `method` that sends a message, set the `imax` variable on the `waitfor` loop with an integer representing the maximum number of participants that require the message.

One-from-many receive. This pattern describes the scenario where a party receives a number of logically related messages that arise from autonomous events occurring at different parties. MAP solution: A `receive (<=)` action could be wrapped by `waitfor` and `timeout` operations. The stop condition of the loop could be set by the number of messages received using the `imax` variable, or by a timing constraint, using the `tmax` variable. If `tmax` is met, i.e., messages are not received within a particular window, compensation actions could be defined within the `timeout` clause. Finally, if the receiving signatures are different for each of the participants, it is possible to define multiple signatures separated by the `or else` operator, as defined in the Racing incoming messages pattern.

One-to-many send/receive pattern. A party sends a request to several other parties. Responses are expected within a given time frame, the interaction may complete successfully or not depending on the set of responses gathered. MAP solution: This pattern is similar to the One-to-many send and One-from-many receive patterns. The temporal constraint is set through the `tmax` value. If the `tmax` value is met without receiving the required number of responses, compensation actions can be defined within the `timeout` clause.

Multiresponses. A party X sends a request to another party Y. Subsequently, X receives any number of responses from Y until no further responses are required. MAP solution: This pattern is directly supported through the `waitfor` loop, with the `imax` variable and corresponding `timeout` operation and `tmax` variable.

Contingent requests pattern. A party X makes a request to another party Y. If X does not receive a response within a certain time frame, X alternatively sends a request to another party Z. MAP solution: A `method m1` is defined which takes as input the name of a participant in the first case Y. A `send (=>)` action is parameterized by the name of the participant (obtained as input to `m1`), once complete a `receive (<=)` action waits for the response from the same participant. The `tmax` variable is set in the `timeout` clause. If the time constraint is met, the `timeout` clause is executed. Within the `timeout` clause, a recursive `call` to `m1` is made using the name of another party, party Z as input, this starts the process again and ignores any further correspondence to party X. Naturally, this can be recursive until the stop condition, i.e., no further participants to send messages to is met.

Atomic multicast notification. A party sends notifications to several parties such that a certain number of parties are required to accept the notification within a certain time frame. For example, all parties or just one party are required to accept the notification, i.e., two-phase commit. MAP solution: This pattern is implementable by defining a `method` which sends the requests, a `method` which listens for positive responses (negative ones can be ignored) within a set time, i.e., by defining `tmax` from a set number of participants, i.e., by defining `imax`. If it fails, i.e., not enough positive responses are received, a compensation `method` sends out abort messages.

Request with referral. Party A sends a request to party B indicating that any follow-up response should be sent to a number of other parties depending on the evaluation of certain conditions. MAP solution: Party A sends a message containing a variable of peer type/role type to party B. Party B then utilizes the peer type/role type information, splicing it into any corresponding `send (=>)` actions, i.e., follow-up responses. This is also known as link passing mobility.

Relayed request pattern. Party A makes a request to party B which delegates the request to other parties $(P1, \ldots, Pn)$. Parties $P1, \ldots, Pn$ then continue interactions with party A, while party B observes a view of the interactions including faults. MAP solution: Passing peer type/role type variables are specified in the Request with referral pattern.

Dynamic routing. A request is required to be routed to several parties based on a routing condition. The routing order is flexible and more than one party can be activated to receive a request. When the parties that were issued the request have completed, the next set of parties pass the request. Routing can be subject to dynamic conditions based on data contained in the original request or obtained in one of the intermediate steps. MAP solution: MAP does not directly support this pattern; however, it is possible to implement a solution based on Web service calls to evaluate the dynamic conditions for the routing.

6 Related Work

This Section discusses all related work from the literature, spanning pure choreography languages, enhancements to widely used modeling techniques, i.e., BPMN, decentralized orchestration, data flow optimization architectures, and Grid toolkits.

6.1 Choreography Languages

There are an overwhelming number of pure orchestration languages, this Section reviews languages targeted specifically at choreography: WS-CDL, Let's Dance, and BPEL4Chor.

The WS-CDL is the proposed standard for service choreography, currently at the W3C Candidate Recommendation stage. However, WS-CDL has met criticism [6], [16] through the Web services community. It is not within the scope of this paper to provide a detailed analysis of the constructs of WS-CDL, this research has already been presented [20]. However, it is useful to point out the key criticisms with the language: WS-CDL choreographies are tightly bound to specific WSDL interfaces, WS-CDL has no multiparty support, no agreed formal foundation, no explicit graphical support, and few or incomplete implementations.

Let's Dance [39] is a language that supports service interaction modeling both from a global and local viewpoint. In a global (or choreography) model, interactions are described from the viewpoint of an ideal observer who oversees all interactions between a set of services. Local models, on the other hand, focus on the perspective of a particular service, capturing only those interactions that

directly involve it. Using Let's Dance, a choreography consists of a set of interrelated service interactions which correspond to message exchanges. Communication is performed by an *actor* playing a *role*. Interaction is specified using one of three Let's Dance constructs: *precedes*; the source interaction can only occur after the target interaction has occurred, *inhibits*; denotes that after the source interaction has occurred, the target interaction can no longer occur, and finally, *weak precedes*; denotes that the target interaction can only occur after the source interaction has reached a final status, e.g., completed or skipped. A complete overview of the Let's Dance language is presented in [39], including solutions to the Service Interaction Patterns.

BPEL4Chor [15] is a proposal for adding an additional layer to BPEL to shift its emphasis from an orchestration language to a complete choreography language. BPEL4Chor is a simple, collection of three artifact types: *participant behavior descriptions* define the control flow dependencies between activities, in particular between communication activities, at a given participant. A *participant topology* describes the structural aspects of a choreography by specifying participant types, participant references, and message links; this serves as the glue between the participant behavior descriptions. Finally, *participant groundings* define the technical configuration details, the choreography becomes Web service specific, concrete links to WSDL definitions, and XSD types are established. BPEL4Chor is an effective proposal and importantly sticks to standards [3] by enhancing the industrially supported BPEL specification. BPEL4Chor encourages reuse by only providing a specific Web service mapping in the participant grounding. Furthermore, unknown numbers of participants can be modeled, not possible with WS-CDL.

6.2 Modeling Support

There are several proposals for extending the Business Process Modeling Notation [2]; the de facto standard for business process modeling. Although the BPMN allows an engineer to define choreographies through a swimlane concept and a distinction between control flow and message flow, it only provides direct support for a limited set of the Service Interaction Patterns and not some of the more advanced choreography scenarios. Decker and Barros [13] introduce a set of extensions for BPMN which facilitate an interaction modeling approach as opposed to modeling interconnected interface behavior models. Authors claim that choreography designers can understand models more effectively, introduce less errors, and build models more efficiently. Evaluation concludes that the majority of the Service Interaction Patterns can be expressed with the additional extensions. Decker and Puhlmann [17] discuss the deficiencies of the BPMN for choreography modeling and proposes a number of direct extensions for the BPMN which overcome these limitations.

6.3 Techniques in Data Flow Optimization

There are a limited number of research papers which have identified the problem of a centralized approach to service orchestration when dealing with data-centric workflows. For completeness, this Section presents an overview of a number of architectures.

The *Circulate* architecture [5] maintains the robustness and simplicity of centralized orchestration, but facilitates choreography by allowing services to exchange data directly with one another. Performance analysis [4] concludes that a substantial reduction in communication overhead results in a 2-4 fold performance benefit across all workflow patterns. An end-to-end pattern through the Montage workflow (a benchmark for the HPC community) results in an 8-fold performance benefit and demonstrates how the advantage of using the architecture increases as the complexity of a workflow grows.

In [31], the scalability argument made in this paper is also identified. The authors propose a methodology for transforming the orchestration logic in BPEL into a set of individual activities that coordinate themselves by passing tokens over shared, distributed tuple spaces. The model suitable for execution is called Executable Workow Networks (EWFN), a Petri nets dialect.

Triana [35] is an open-source problem solving environment. It is designed to define, process, analyze, manage, execute, and monitor workflows. Triana can distribute sections of a workflow to remote machines through a connected peer-to-peer network. *OGSA-DAI [26]* middleware supports the exposure of data resources on to Grids and facilitates data streaming between local OGSA-DAI instances. *Grid Services Flow Language (GSFL) [29]* addresses some of the issues discussed in this paper in the context of Grid services, in particular, services adopt a peer-to-peer data flow model.

7 Conclusions

The motivating Redshift scenario, taken from the AstroGrid science use-cases, demonstrated how centralized orchestration can become a bottleneck to the performance of a workflow, extra copies of data are sent that consume network bandwidth and overwhelm the central engine. Choreography does not rely on centralization, as a result, services can pass data directly to where they are required, at the next service in the workflow. We argue that as the number of services and the size of data involved in workflows increase, traditional centralized orchestration techniques are reaching the limits of scalability. Choreography techniques, although more complex to model offer a decentralized alternative and for data-centric workflows, are the optimal architecture.

This paper has introduced the MAP choreography language, an implementation of interconnection choreography models. Our aim was to demonstrate how service choreographies can be specified, verified, and enacted with a comparatively simple process language, with a focus on data flow optimization. The MAP language syntax, corresponding open-source implementation and model checking environment have been discussed in the context of our motivating Redshift scenario. MAP was evaluated by demonstrating the languages conformance to the Service Interaction Patterns and through use-case, by implementing the Redshift scenario. Returning to our original aims in Section 1.3, we highlight the contributions this paper has made with reference to the Related Work, discussed in Section 6.

7.1 Loosely Coupled Choreography Interface

Peers are decoupled from the Web services they invoke, i.e., they are an entirely separate component. This decoupling means that Web services need no alteration or knowledge that they are even taking part in a choreography. Therefore, no modification of Web services needs to take place prior to enactment; although for optimal data flow, a peer needs to be installed as closely as possible to the Web service(s) it is to invoke so that communication goes over a Local Area, not Wide Area network. Web services are owned and maintained by different organizations and may not agree to installing specialist interfaces in order to facilitate choreography, unless they have something to gain. Peers are less intrusive and offer an advantage as they are entirely external to the Web services.

In comparison, WS-CDL is invasive to the Web services themselves as an engineer must agree to and program a closely coupled WS-CDL interface (which sits above WSDL) in order to take part in a choreography. It is entirely possible for WS-CDL and other choreography languages to be decoupled from the Web services they make use of; however, MAP has introduced this architecture and provides this functionality by default.

7.2 Executable Choreography Language

This paper has introduced the notion of an executable choreography language. Peers act as blank canvases and do not have to be preconfigured with a particular choreography definition in advance. Instead a choreography, specified as a MAP protocol, can be sent to a group of peers across a network, each peer dynamically reads in the MAP protocol, assumes a role within the protocol, and enacts the choreography across the set of distributed peers. In contrast, an engineer must preinstall and configure existing choreography languages at design-time in order for a service to take part in a choreography. The MAP approach has been applied to live e-Science scenarios in the Astronomy domain: first to the UK e-Science project AstroGrid and second to the Large Synoptic Survey Telescope project (LSST). These applications have demonstrated the use of MAP to implement real-world scenarios where large data sets are passed between collaborating peers.

We have demonstrated that although MAP does not have dedicated constructs for dealing with the Service Interaction Patterns, 12 of the 13 patterns can be implemented by combining actions and operations together in novel ways. MAP supports multiparty communication between an unknown and unbounded set of participants, allowing multiple MAP protocols to be executed concurrently. These multiparty scenarios are very common, as demonstrated by the Service Interaction Patterns. In contrast, WS-CDL does not directly support parallel conversations with an unknown number of participants. That means, all participants have to be modeled before the enactment of a choreography and makes auctions scenarios with an unknown number of bidders impossible. Unknown numbers of participants are natively supported in BPEL4Chor [15].

7.3 Verification through Model Checking

Asynchronous service choreography introduces nondeterminism into the system which causes a number of potential problems such as synchronization, fairness, and deadlocks. Model checking techniques, discussed in Section 3.4, have been introduced in order to verify certain properties of a choreography (e.g., termination) before it is deployed live over a network. In contrast, WS-CDL borrows terminology from π-calculus, though there are no accepted formal semantics, an early attempt is made here [23]. In [30], authors demonstrate how BPEL4Chor can be verified through Petri nets. Let's Dance also has documented execution semantics [19] for the language in terms of a mapping to π-calculus; these formal semantics provide a basis for analyzing choreographies.

7.4 Open-Source Implementation

Choreography modeling techniques (e.g., extensions to BPMN), discussed in Section 6.2, are useful for designers to build models more efficiently; however, they provide no framework for enactment. Importantly, MAP is not a simulation environment, the MAP specification has been implemented as an open-source Web services choreography tool kit, MagentA, which provides a concrete API and framework for the enactment of distributed choreographies. The importance of concrete implementations is often overlooked in the literature.

At the time of writing, there are only two documented, prototype implementations of the WS-CDL specification. The work of WS-CDL+ [24] proposes six extensions to the WS-CDL specification to enhance expressiveness and usability. The extended specification [25] has been implemented in prototype form, although only one version, version 0.1, has been released. A further partial implementation [20] of the WS-CDL specification is currently in the prototype phase. The other widely known implementation is `pi4soa`,[5] an Eclipse plugin which provides a graphical editor to compose WS-CDL choreographies and generate from them compliant BPEL. Maestro [14] is an implementation of the Let's Dance language and supports the static analysis of global models, the generation of local models from global ones, and the interactive simulation of both local and global modes. In the BPEL4Chor space, A Web-based editor[6] allows engineers to graphically build choreography models.

7.5 Future Work

Research based on the MAP approach to service choreographies is ongoing through the Open Knowledge project[7] an European Union FP6 project. The OpenKnowledge project aims at knowledge sharing through open and flexible peer interactions. Within this project, the members are developing a system that supports searching, developing, and sharing of interactions/workflows consisting of roles implemented by software that can be shared and executed by peers. Its main requirements are openness, scalability, decentralization, and robustness.

5. http://sourceforge.net/projects/pi4soa [16/12/2008].
6. http://www.bpel4chor.org/editor/ [16/12/2008].
7. http://www.openk.org/ [16/12/2008].

REFERENCES

[1] Interaction Protocol Specifications, technical report, Foundation for Intelligent Physical Agents, http://www.fipa.org/repository/ips.php3, 2002.

[2] Business Process Modeling Notation (BPMN) Specification, technical report, Object Management Group (OMG), http://www.omg.org/spec/BPMN/1.1/PDF, 2006.

[3] A. Barker and J. van Hemert, "Scientific Workflow: A Survey and Research Directions," *Proc. Seventh Int'l Conf. Parallel Processing and Applied Math., Revised, Selected Papers,* R. Wyrzykowski et al., eds., pp. 746-753, 2008.

[4] A. Barker, J.B. Weissman, and J. van Hemert, "Eliminating the Middle Man: Peer-to-Peer Dataflow," *Proc. 17th Int'l Symp. High Performance Distributed Computing (HPDC '08),* pp. 55-64, 2008.

[5] A. Barker, J.B. Weissman, and J. van Hemert, "Orchestrating Data-Centric Workflows," *Proc. Eighth IEEE Int'l Symp. Cluster Computing and the Grid (CCGrid '08),* pp. 210-217, 2008.

[6] A. Barros, M. Dumas, and P. Oaks, "A Critical Overview of the Web Services Choreography Description Language (WS-CDL)," *BPTrends Newsletter 3,* 2005.

[7] A. Barros, M. Dumas, and A. ter Hofstede, "Service Interaction Patterns," *Proc. Third Int'l Conf. Business Process Management,* pp. 302-318, 2005.

[8] M. Benerecetti, F. Giunchiglia, and L. Serafini, "Model Checking Multiagent Systems," *J. Logic and Computation,* vol. 8, no. 3, pp. 401-423, 1998.

[9] E. Bertin and S. Arnouts, "Sextractor: Software for Source Extraction," *Astronomy and Astrophysics Supplement Series,* vol. 117, pp. 393-404, 1996.

[10] R.H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge, "Model Checking AgentSpeak," *Proc. Second Int'l Conf. Autonomous Agents and Multiagent Systems (AAMAS '03),* 2003.

[11] R.H. Bordini, M. Fisher, W. Visser, and M.J. Wooldridge, "State-Space Reduction Techniques in Agent Verification," *Proc. Third Int'l Conf. Autonomous Agents and Multiagent Systems (AAMAS '04),* pp. 896-903, 2004.

[12] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking.* MIT Press, 1999.

[13] G. Decker and A. Barros, "Interaction Modeling Using BPMN," *Proc. First Int'l Workshop Collaborative Business Processes (CBP '07),* pp. 206-217, 2007.

[14] G. Decker, M. Kirov, J.M. Zaha, and M. Dumas, "Maestro for Let's Dance: An Environment for Modeling Service Interactions," *Proc. Demonstration Session of the Fourth Int'l Conf. Business Process Management (BPM '06),* 2006.

[15] G. Decker, O. Kopp, F. Leymann, and M. Weske, "BPEL4Chor: Extending BPEL for Modeling Choreographies," *Proc. IEEE Int'l Conf. Web Services (ICWS '07),* pp. 296-303, 2007.

[16] G. Decker, H. Overdick, and J.M. Zaha, "On the Suitability of WS-CDL for Choreography Modeling," *Proc. Methoden, Konzepte und Technologien für die Entwicklung von Dienstebasierten Informationssystemen (EMISA '06),* pp 21-34, 2006.

[17] G. Decker and F. Puhlmann, "Extending BPMN for Modeling Complex Choreographies," *Proc. 15th Int'l Conf. Cooperative Information Systems (CoopIS '07),* pp. 24-40, 2007.

[18] G. Decker and M. Weske, "Local Enforceability in Interaction Petri Nets," *Proc. Fifth Int'l Conf. Business Process Management,* pp. 305-319, 2007.

[19] G. Decker, J.M. Zaha, and M. Dumas, "Execution Semantics for Service Choreographies," *Proc. Third Int'l Workshop Web Services and Formal Methods (WS-FM '06),* pp. 163-177, 2006.

[20] L. Fredlund, "Implementing WS-CDL," *Proc. Second Spanish Workshop Web Technologies (JSWEB '06),* 2006.

[21] D. Hollingsworth, *The Workflow Reference Model.* Workflow Management Coalition, 1995.

[22] G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual.* Addison Wesley, 2003.

[23] Z.Y. Hongli and Q. Zongyan, "A Formal Model of the Web Service Choreography Description Language (WS-CDL)," technical report, Dept. of Informatics, Peking Univ., 2006.

[24] Z. Kang, W. Hongbing, and P.C. Hung, "WS-CDL+ for Web Service Collaboration," *Information Systems Frontiers,* vol. 9, no. 4, pp. 375-389, 2007.

[25] Z. Kang, H. Wang, and P.C. Hung, "WS-CDL+: An Extended WS-CDL Execution Engine for Web Service Collaboration," *Proc. IEEE Int'l Conf. Web Services (ICWS '07),* pp. 928-935, 2007.

[26] K. Karasavvas, M. Antonioletti, M. Atkinson, N.C. Hong, T. Sugden, A. Hume, M. Jackson, A. Krause, and C. Palansuriya, *Introduction to OGSA-DAI Services,* vol. 3458, pp. 1-12, 2005.

[27] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon, *Web Services Choreography Description Language (WS-CDL) Version 1.0.* W3C Candidate Recommendation, 2005.

[28] S. Kotoulas and R. Siebes, "Adaptive Routing in Structured Peer-to-Peer Overlays," *Proc. Third Int'l IEEE Workshop Collaborative Service-Oriented P2P Information Systems (Convective and Orographically-induced Precipitation Study (COPS) Workshops Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE '07)),* 2007.

[29] S. Krishnan, P. Wagstrom, and G.V. Laszewski, "GSFL: A Workflow Framework for Grid Services," technical report, Argonne Nat'l Laboratory, 2002.

[30] N. Lohmann, O. Kopp, F. Leymann, and W. Reisig, "Analyzing BPEL4Chor: Verification and Participant Synthesis," *Proc. Fourth Int'l Workshop Web Services and Formal Methods (WS-FM '08),* M. Dumas and R. Heckel, eds., pp. 46-60, 2008.

[31] D. Martin, D. Wutke, and F. Leymann, "A Novel Approach to Decentralized Workflow Enactment," *Proc. 12th Int'l IEEE Conf. Enterprise Distributed Object Computing (EDOC '08),* pp. 127-136, 2008.

[32] R. Milner, J. Parrow, and W. David, "A Calculus of Mobile Processes Pt.1," *Information and Computation,* vol. 100, no. 1, pp. 1-77, 1992.

[33] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, and P. Li, "Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows," *Bioinformatics,* vol. 20, pp. 3045-3054, 2004.

[34] S. Ross-Talbot, "Orchestration and Choreography: Standards, Tools and Technologies for Distributed Workflows," *Proc. Workshop Network Tools and Applications in Biology (NETTLAB '05),* 2005.

[35] I. Taylor, M. Shields, I. Wang, and R. Philp, "Distributed P2P Computing within Triana: A Galaxy Visualization Test Case," *Proc. 17th Int'l Parallel and Distributed Processing Symp. (IPDPS '03),* pp. 16-27, 2003.

[36] The OASIS Committee, *Web Services Business Process Execution Language (WS-BPEL) Version 2.0.* 2007.

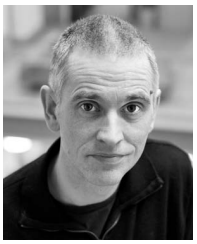
[37] C.D. Walton, "Verifiable Agent Dialogues," *J. Applied Logic,* vol. 5, no. 2, pp. 197-213, 2007.

[38] M. Wooldridge, M. Fisher, M.P. Huget, and S. Parsons, "Model Checking Multiagent Systems with MABLE," *Proc. First Int'l Conf. Autonomous Agents and Multiagent Systems (AAMAS '02),* 2002.

[39] J.M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede, "Let's Dance: A Language for Service Behavior Modeling," *Proc. On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated Int'l Confs.,* R. Meersman and Z. Tari, eds., pp. 145-162, 2006.

Adam Barker received the BSc degree in software engineering and the MSc degree in distributed systems from Newcastle University. He received the PhD degree in informatics from the University of Edinburgh. He is currently a postdoctoral research associate in the Department of Engineering Science, University of Oxford, United Kingdom. Prior to joining Oxford, he was employed as a postdoctoral research associate at the National e-Science Centre (NeSC). Complementing his academic experience, he has completed internships at Hewlett-Packard and BAe Systems. His primary research interests concentrate on the effective engineering of large-scale, distributed systems, operating in open, decentralized, and uncertain environments. His research agenda is pursued by modeling, evaluating, and building novel architectures and algorithms based on sound foundations in systems research. For further information, please refer to www.adambarker.org.

Christopher D. Walton received the BSc degree in computer science from the University of Edinburgh in 1996, and the PhD degree from the School of Informatics at the University of Edinburgh in 2001. He is currently the lead researcher at Metaforic, Ltd., a software security company. His research interests there include reasoning about software security, knowledge dissemination using Semantic Web technologies, and the implementation of secure Web-based computation. He is a author of the book entitled *Agency and the Semantic Web* (Oxford Univ. Press, 2006). He was previously a researcher on the EU-funded Open Knowledge (www.openk.org) and EPSRC-funded Advanced Knowledge Technologies (www.aktors.org) research programmes.



David Robertson is the director of the Centre for Intelligent Systems and their Applications, part of the School of Informatics, at the University of Edinburgh. His current research is on formal methods for coordination and knowledge sharing in distributed, open systems—the long-term goal being to develop theories, languages, and tools that outperform conventional software engineering approaches in these arenas. He was a coordinator of the OpenKnowledge project (www.openk.org) and was a principal investigator on the Advanced Knowledge Technologies research consortium (www.aktors.org), which are major EU and UK projects in this area. His earlier work was primarily on program synthesis and the high-level specification of programs, where he built some of the earliest systems for automating the construction of large programs from domain-specific requirements.