Typed Protocols for Peer-to-Peer Service Composition

Christopher D. Walton*
Centre for Intelligent Systems and their Applications (CISA),
School of Informatics, University of Edinburgh, Scotland, UK.
Email: cdw@inf.ed.ac.uk

Abstract

In this paper we present a technique which addresses the composition of web services into peer-to-peer systems. Our approach is founded on the definition of lightweight protocols, which provide the means to specify, execute, and verify these systems. The advantage of our approach is that the protocols are defined independent of the domain in question, and therefore allow us to focus specifically on the composition aspects of the system. We present a definition of the MAP language for service composition, and show how it can be used to specify a simple peer-to-peer file-sharing system. We also illustrate how the use of type information can allow us to gain confidence in the correctness of our protocols.

1 Introduction

A peer-to-peer (P2P) architecture is one which allows autonomous peers of similar capabilities to interact in a distributed and decentralised manner. The advantage of the P2P approach, over a centralised client/server architecture, is that the network resources are effectively utilised, yielding more scalable and robust operation. P2P architectures have recently gained significant popularity for the distribution of files over the Internet. However, the potential scope for P2P techniques is much greater, and they can be effectively used in a range of different domains, including the Semantic Web, Grid Computing, Database Systems, and Multi-Agent Systems.

The P2P approach encompasses many different techniques from service-oriented and agent-oriented architectures. However, in a P2P system, these techniques are applied in a purpose-driven manner. That is, a P2P system is focused on achieving a specific task, and typically within some reasonable deadline. Therefore, techniques which rely on undecidable reasoning, or lengthy theorem-proving operations are unsuitable for P2P systems. Instead, P2P systems rely on decidable and practical reasoning techniques which can be straightforwardly utilised by distributed peers. Nonetheless, there remain significant challenges to be addressed in the construction of P2P architectures, particularly in relation to the composition and inter-operability of peers.

For convenience, we make the assumption that the peers in a P2P architecture are represented and controlled though a *web service* interface. It is intended that P2P systems will be rapidly constructed by combining a range of different services. Thus, the construction of a P2P architecture is essentially that of web service *composition*. The effective composition of services into a P2P systems is the theme of this paper. The composition of web services into P2P architectures, requires a high degree of *interoperability* between services. It is necessary that services built by different organisations, and using different software systems, are able to communicate with one another in a common formalism with an agreed semantics.

Interaction between web services is currently accomplished by a remote procedure call (RPC) mechanism, by the exchange of SOAP messages between web service clients and containers. The SOAP specification defines a one-way stateless communication mechanism, but this is too restrictive for our purposes. In order to compose services, we need to define complex communication patterns between services, for example, broadcast or multi-cast communication. It is possible to construct such a system through a static composition of services, where the composition is encoded directly into the services. However, this approach is error-prone and

*This work is sponsored by the EPSRC Advanced Knowledge Technologies (AKT) Interdisciplinary Research Collaboration (Grant GR/N15764/01).

inflexible as it does not allow us to easily change the kind of system we define. Ideally, we would like a separate representation which can express complex patterns of interaction between web services, such as we describe. We are aware of the activities of the W3C Web Services Choreography group on defining a suitable representation for performing choreography (i.e. composition) between web services [3], but we are unaware of any implementation of these proposals.

From our discussion, we would like to specify a tightly-coupled predictable system with complex communication behaviour and a high degree of interoperability. The approach to service composition which we adopt in this paper is a dynamic approach, where services can be composed in a flexible manner with recursive and concurrent behaviours. As stated previously, there is some overlap between agent architectures and P2P architectures. In particular, there are many conceptual similarities between agents and peers [6, 8]. We use this similarity to our advantage in this paper, as the composition techniques that we present are directly adapted from our previous work on interaction protocols in Multi-Agent Systems [12, 10].

The focus of this paper is on the definition of a formalism for the realisation of interaction between web services in a P2P architecture. We present a script-based representation for the interaction between services, which is lightweight and verifiable. Our language appears to be a good complement to SOAP, as both are independent of the message content or implementation. Our approach has some similarities to the work in [7] which defines a formalism based on petri-nets for coordinating BDI agents in a P2P architecture. However, our approach is based on process calculus [4], and is not restricted to one specific model of agency. Our work is also similar to the representation of agent interactions found in Electronic Institutions [2], though our specifications are designed to be directly executable. We also address the verification of protocols, by defining a formal type-system for our language.

Our presentation in this paper is structured as follows. In section 2 we define the MAP language for specifying an enacting protocols. To demonstrate the key features of the language, we present the specification of an example P2P file-sharing protocol in Section 3. The version of MAP presented in this paper has been extended with type information, which we use to perform type-based verification of MAP protocols in Section 4. Lastly, we conclude in Section 5 with a discussion of our future work.

2 MAP Language Definition

The MAP protocol language which we present here is a lightweight protocol language derived from process calculus, specifically the π calculus [5]. MAP is also derived from our previous work on multi-agent protocols, and thus we will use the terms *peer* and *agent* interchangeably. MAP protocols can be viewed as *executable specifications*, and we have defined an execution framework for MAP, called MagentA [11]. Two key concepts in MAP are the division of protocols into *scenes*, and the assignment of *roles* to the peers. A scene can be thought of as a bounded space in which a group peers interact on a single task. Thus, a scene divides a large protocol into manageable parts. Scenes also add a measure of security to a protocol, in that peers which are not relevant to the protocol are excluded from the scene. This can prevent interference with the protocol and limits the number of exceptions and special cases that must be considered in the design of the protocol. We assume that a scene places a barrier on the peers, such that a scene cannot begin until all the peers have been instantiated.

The concept of a *role* is also central to our definition. In MAP, each peer is identified by both a name and a role. Peers are uniquely named, but must be assigned a role which is specified in the protocol. The role of an peer is fixed until the end of a scene, 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 web service interface. Roles are useful for grouping similar peers together, as we do not have to specify a completely separate protocol for each individual. For example, we may wish to interact with a large number of services, all with the same interface. We can simply define a single role (and associated protocol) which corresponds to the interface, rather than defining a separate protocol for each service. Roles also allow us to specify multi-cast communication in MAP. For example, we can broadcast messages to all peers of a specific role.

We note that MAP is only intended to express protocols, and is not intended to be a general-purpose programming language. Therefore, the relative lack of features for performing computation is appropriate. Furthermore, MAP is designed to be a lightweight language and only a minimal set of operations have been included. It is intended that MAP protocols will be automatically generated, e.g. from a planning system. Thus, although MAP protocols appear complex, they

would not generally be constructed by hand.

We will now define the abstract syntax of MAP, which is presented in Figure 1 (BNF notation). We have also defined a corresponding concrete XML-based syntax for MAP which is used in our implementation. However, we restrict our attention in this paper to the abstract syntax for readability. A protocol P is uniquely named n and defined as a set of roles r, each of which defines a set of methods $\mathcal{M}$. A method m takes a list of terms $\phi^{(k)}$ as arguments (the initial method is named `main`). Agents (i.e. peers) have a fixed role r for the duration of the protocol, and are individually identified by unique names a. Protocols are constructed from operations op which control the flow of the protocol, and actions α which have side-effects and can fail. Failure of actions causes backtracking in the protocol.

P	::=	$n(r\{\mathcal{M}\})^+$	(Scene)
M	::=	$\texttt{method}\ m(\phi^{(k)}) = op$	(Method)
op	::=	α	(Action)
	|	$op_1\ \texttt{then}\ op_2$	(Sequence)
	|	$op_1\ \texttt{or}\ op_2$	(Choice)
	|	$\texttt{waitfor}\ op_1\ \texttt{timeout}\ op_2$	(Iteration)
	|	$\texttt{call}\ m(\phi^{(k)})$	(Recursion)
α	::=	ϵ	(No Action)
	|	$\phi^k = p(\phi^l)\ \texttt{fault}\ \phi^m$	(Procedure)
	|	$\rho(\phi^{(k)})\ \texttt{=>}\ \texttt{agent}(\phi_1,\ \phi_2)$	(Send)
	|	$\rho(\phi^{(k)})\ \texttt{<=}\ \texttt{agent}(\phi_1,\ \phi_2)$	(Receive)
ϕ	::=	$_ \mid a \mid r \mid c : \tau \mid v : \tau$	
τ	::=	$utype \mid atype \mid rtype \mid tname$	

Figure 1: MAP abstract syntax.

The interface between the protocol and the web service which defines its behaviour, is achieved through the invocation of procedures p. A procedure is parameterised by three sequences of terms. The input terms ϕ^l are the input parameters to the procedure, and the output terms ϕ^k are the output parameters, i.e. results, from the procedure. A procedure may also raise an exception in which case the fault terms ϕ^m are bound to the exception parameters, and backtracking occurs in the protocol. Interaction between agents is performed by the exchange of messages which are defined by performatives ρ, i.e. message types. The parameters to procedures and performatives are terms ϕ, which are either variables v, agent names a, role names r, constants c, or wild-cards _. Literal data is represented by constants c in our language, which can be complex data-types, e.g. currency, flat-file data, multimedia, or XML documents. Variables are bound to terms by unification which occurs in the invocation of procedures, the receipt of messages, or through recursive method invocations. Constants and variables are assigned explicit types τ to ensure that they are treated consistently. We present a formal type-system in Section 4, which is a complement to our formal semantics, previously defined in [10].

3 Example Scenario

It is helpful to consider an example scenario in order to obtain an understanding of the MAP protocol language. Our example is based on a simplified peer-to-peer (P2P) file sharing system. This kind of system has recently gained significant popularity for the decentralised distribution of multimedia files on the Internet, e.g. mp3 music. At present, the majority of file-sharing P2P systems are based on purely algorithmic techniques. However, this kind of system appears ideal for the use of agent-based technology. For example, we can readily anticipate the exchange of files through negotiation between agents.

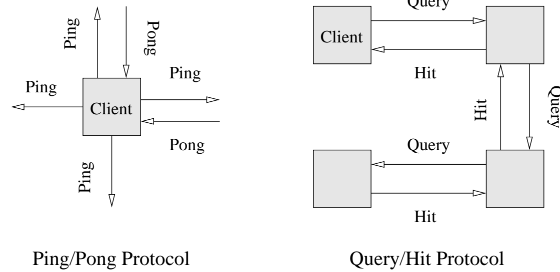

Figure 2: Query flooding protocols.

The model that we describe is loosely based on the Gnutella file sharing protocol. This protocol defines a completely decentralised method of file sharing, and its implementation is very straightforward. The Gnutella system assumes a distributed network of nodes, i.e. computers, that are willing to share files on the network. The protocol is defined with respect to our own node on the network, which we call the client. There are just three main operations performed by a client in the Gnutella protocol:

1. In order to participate in file sharing it is necessary to locate at least one other active node in the

Gnutella network. There are a variety of ways in which this can be accomplished. The most common way is to contact one of more *Gwebcache* servers which which contain lists of recently active nodes. The Gnutella software is usually preconfigured with the addresses of a large number of these servers. However, it is not enough simply to know about other nodes, as there are no guarantees that these nodes will still be active. Therefore, the client will initiate a simple ping/pong protocol with each node in the list until a certain quota of active nodes have been located. This protocol simply sends a message (ping) to each node in the list, and waits for a certain period of time until a reply message (pong) is received, indicating that the node is still active.

2. Once a list of active nodes has been obtained, it is possible to perform a search for a particular file. Gnutella uses a *query flooding* protocol to locate files on the network. The client sends the file request (query) to every node on its active list. If one of the nodes has a copy of the requested file, then it sends back a reply message (hit) to the client. If the node does not have the file, then the request is forwarded to all of the active nodes on its own list, and so on. The query will eventually propagate to all of the nodes on the network, and the reply will be returned to the client.

3. If the file is successfully located by the query protocol, then then client simply contacts the destination node directly and initiates the download. If more than one copy is located, the client may download fragments of the file from different locations simultaneously and thereby improve download performance.

The Gnutella *ping/pong* and *query/hit* protocols are illustrated in Figure 2. It should be noted that the basic query flooding protocol, as outlined here, is very inefficient in operation and a search will typically take a long time to complete. The number of messages required is exponential in the depth of the search. This behaviour is tolerated as the network traffic generated by the queries is very small, compared to the bandwidth required to transfer the file itself. A variety of caching strategies have been proposed to improve the speed of the search, though we only consider the basic protocol here.

We can readily express the Gnutella protocol in MAP. The agents follow the protocol to determine the actions that must be performed by the nodes to retrieve a particular file. The encoding is presented in Figure 3 for the file-sharing nodes, and Figure 4 for an external client. We distinguish between the different types of terms by prefixing variables names with `$`, and role names with `%`. We use type abbreviations `a` for an agent, `r` for role, and `alist` for a list of agents.

The protocol for a node, shown in Figure 3, proceeds as follows. Upon initialisation (line 3), a list of neighbouring nodes is obtained, and a ping message is sent to all of these nodes in turn (lines 5-8). The node then enters a responsive state where is listens for incoming messages, and acts according to the message type. An incoming ping message results in an outgoing pong message (line 11). An incoming pong message is recorded in the list of active nodes (line 12). An incoming query results in an outgoing hit message if the node has a copy of the file (lines 13-15), or the query is forwarded to all of the neighbouring nodes (lines 16-18). An incoming hit message (from a neighbour) is forwarded to the initial requester of the query (lines 19-21). Finally, a download request message results in an outgoing message containing a copy of the file (lines 22-24). The protocol repeats after each message (line 25). For brevity, the `sendquery` and `sendhits` methods have been omitted as they have a similar definition to the `sendping` method. The procedures `startSharing`, `addActive`, `getActiveNodes`, `recordQuery`, `getQueryList`, and `getFile` are defined in the web service associated with each node, as they are external to the composition process. The client protocol shown in Figure 4 interacts directly with the node protocol that we have defined. A client obtains a node on the network (line 3), and constructs a query (line 4). The query is forwarded to the node (line 5), and the client waits for a hit message to be returned (line 6). A download is then initiated from the node which has a copy of the file (lines 7-8).

Our MAP protocols are clearly a straightforward implementation of the required functionality. However, there are some subtle issues that require further explanation. The operations in the protocol are sequenced by the `then` operator which evaluates op_1 followed by op_2, unless op_1 involved an action which failed. The failure of actions is handled by the `or` operator. This operator is defined such that if op_1 fails, then op_2 is evaluated, otherwise op_2 is ignored. The language includes backtracking, such that the execution will backtrack to the nearest `or` operator when a failure occurs.

```
1 %node{
2   method main() =
3     $id:a = getId() then startSharing($id:a) then $nodes:alist = getNodes() then
4     (call sendping($nodes:alist) or call mainloop($id:a))
5   method sendping($nodes:alist) =
6     $head:a = Head($nodes:alist) fault nohead then
7     $tail:alist = Tail($nodes:alist) fault notail then
8     ping() => agent($head:a, %node) then call sendping($tail:alist)
9   method mainloop($id:a) =
10    waitfor
11         ((ping() <= agent($n:a, %node) then pong() => agent($n:a, %node))
12      or ((pong() <= agent($n:a, $role:r) then addActive($n:a, $role:r))
13      or ((query($f:string) <= agent($n:a, $r:r) then
14           ($fl:file = getFile($f:string) fault nofile then
15            hit($f:string, $id:a) => agent($n:a, $r:r))
16        or (setQuery($f:string, $n:a, $r:r) then
17            $nodes:alist = getActiveNodes() then
18            call sendquery($f:string, $nodes:alist)))
19      or ((hit($f:string, $hid:a) <= agent($n:a, %node) then
20          $nodes:alist = getQueryList($f:string) then
21          call sendhits($f:string, $hid:a, $nodes:alist))
22      or (download($f:string) <= agent($client:a, %client) then
23          $fl:file = getFile($f:string) fault nofile then
24          file($fl:file) => agent($client:a, %client))))))
25    then call mainloop($id:a)}
```

Figure 3: MAP encoding of a P2P node.

Similarly, the body of a `waitfor` loop will be repeatedly executed upon failure, and the loop will terminate when the body succeeds.

```
1 %client{
2   method main() =
3     $node:id = getStartNode() then
4     $fname:string = getQuery() then
5     query($fname:string) =>
        agent($node:a, %node) then
6     waitfor (hit($fname:string, $hitid:a)
        <= agent($name:a, $role:r))
7       then download($fname:string) =>
          agent($hitid:a, %node) then
8       waitfor (filereply($file:file) <=
          agent($hitid:a, %node))}
```

Figure 4: MAP encoding of a P2P client.

The semantics of message passing in MAP corresponds to non-blocking, reliable, and buffered communication. Sending a message will succeed immediately if an agent matches the definition, and the message will be stored in a buffer on the recipient. When exchanging messages through send and receive actions, a unification of terms against the definition $\mathtt{agent}(\phi_1,\ \phi_2)$ is performed, where ϕ_1 is matched against the agent name, and ϕ_2 is matched against the agent role. For example, the receipt of the ping message in line 11 of the node protocol will match any agent whose role is `%node`, and the name of this node will be bound to the variable `$n`. In this definition, a client is not permitted to send a ping message to a node. Although not illustrated in this example, we can use a wildcard _ to send a message to all agents regardless of their role. The advantage of non-blocking communication is that we can check for a number of different messages at the same time. Race conditions are avoided by wrapping all receive actions by `waitfor` loops. A `waitfor` loop can also include a `timeout` condition which is triggered after a certain interval has elapsed.

4 Protocol Verification

MAP protocols specify complex, concurrent, and asynchronous patterns of communication. The presence of concurrency introduces *non-determinism* which gives rise to a large number of potential problems, such as synchronisation, fairness, and deadlocks. It is difficult, even for an experienced designer, to obtain a good intuition for the behaviour of a concurrent protocol. This is primarily due to the large number of possible interleavings which can occur, even when considering very

simple protocols. Traditional debugging and simulation techniques cannot readily explore all of the possible behaviours of such systems, and therefore significant problems can remain undiscovered. The detection of problems in these systems is typically accomplished through the use of *formal verification* techniques such as theorem proving and model checking. We have previously shown [9] how we can use the SPIN model checker to verify properties of MAP protocols.

The application of model checking to protocol is a powerful technique for ensuring that certain properties of the protocols hold. However, while model checking itself is fully automatic, it is still necessary to interpret the outcome of the process. In many cases, it can be difficult to pin-point the exact location of the problem. This is principally because the model checking is performed on an encoding of the protocol, rather than directly on the MAP language itself. Therefore, we have recently turned our attention toward improving this process. We note that a significant cause of failure in MAP protocols can be attributed to the pattern matching process. Both method calls, and message passing operations involve a unification step which can fail. Our model checking technique can detect this kind of failure. However, we note that failure of unification is normally detected by a *type checking* process. Thus, our intention is to detect type errors statically before model checking or enactment.

We have recently introduced types into the MAP language, defined by τ in Figure 1. Types are directly assigned to all variables and constants. We define four different types: $utype$ is an unknown (wildcard) type, $atype$ corresponds to an agent name, $rtype$ which corresponds to a role, and $tname$ is a type name. Type names $tname$ will typically be XML schema types, e.g. `xsd:string`. As MAP does not attempt to perform any computation on the constants, our choice of type names is only dependent on the kinds of data that the external web services can interpret. Type checking ensures that these types are used consistently.

Type checking is performed by defining a formal *type system*, from which the checking algorithm can be derived. Our type system is presented in the style of [1], and complements the formal semantics of MAP defined in [10]. A type system is defined by a collection of rules which are used to determine if a program is *well-typed*. Well-typing corresponds to a notion of predictability of behaviour. Our typing rules check for three kinds of consistency in a protocol: every `call` operation matches at least one `method` in the protocol; each procedure p is invoked consistently; and every message ρ is constructed consistently.

Environment	Γ	$::=$	$(RE,\ PE,\ ME)$		
Roles	RE	$::=$	r	$\overset{map}{\longmapsto}$	$\{\tau^{(k)}\}$
Procedures	PE	$::=$	p	$\overset{map}{\longmapsto}$	$(\tau_{in}^{(k)},\ \tau_{out}^{(l)},\ \tau_{flt}^{(m)})$
Messages	ME	$::=$	ρ	$\overset{map}{\longmapsto}$	$\tau^{(k)}$

Figure 5: Typing environment.

Our typing rules, are all defined with respect to a typing environment Γ, illustrated in Figure 5. The type environment is represented by a three-tuple comprising a role environment RE, a procedure environment PE, and a message environment ME. The role environment maps role names r to sets of type sequences $\tau^{(k)}$. Type sequences are the argument types for the methods of the protocol, associated with each role. The procedure environment maps procedure names p to tuples, which defines the types of the input, output, and fault terms for the procedure. Finally, the message environment maps performatives ρ to a sequence of types $\tau^{(k)}$, which are the types of the terms used in the body of the message. We use the abbreviation $\Gamma \cup \{\rho \mapsto \tau^{(k)}\}$ to denote the environment ME in Γ extended with a mapping from ρ to $\tau^{(k)}$, and $\Gamma(\rho) = \emptyset$ indicate that ME in Γ contains no mapping for ρ.

The types of the terms in MAP are defined in Figure 6 (1). The term typings all have the form $\vdash \phi : \tau$, where ϕ is a MAP term, and τ is the resulting type. The unknown type $utype$ is assigned to a wild-card term. The types $aterm$ and $rterm$ are assigned to agent names and role names respectively. Constants and variables are typed according to their definitions.

At the core of the typing process is a very simple unification procedure, defined in Figure 6 (2). As stated earlier, MAP does not attempt to assign any meaning to the types. Thus, the unification rules simply check that the types match. The first case states that any type τ matches an unknown type $utype$. The second case states that two types match if they are identical. The final case states that two sequences of types will match, if they are the same length, and all the individual types in the sequence can be matched.

The MAP typing rules are defined in Figure 6(3-11). The rules are all judgements of the form $\Gamma \vdash \theta : \Gamma'$, which assert that the MAP program fragment θ is valid (i.e. type-able) in Γ, and yields the new environment Γ'. The typing rules are applied recursively, reading from

(1) $\vdash _ : utype \quad \vdash a : atype \quad \vdash r : rtype$
$\vdash (c : \tau) : \tau \quad \vdash (v : \tau) : \tau$

(2) $\vdash unify(utype,\ \tau) \quad \vdash unify(\tau,\ \tau)$
$\vdash unify({\tau_1}^{(k)},\ {\tau_2}^{(k)}) =$
$unity(\tau_1^1,\ \tau_2^1)\ \cdots\ unify(\tau_1^k,\ \tau_2^k)$

(3)
$$\frac{\Gamma,\ r_1 \vdash \mathcal{M}_1 : \Gamma_1\ \cdots\ \Gamma_{n-1},\ r_n \vdash \mathcal{M}_n : \Gamma_n}{\Gamma \vdash n(r_1\{\mathcal{M}_1\},\ \ldots,\ r_n\{\mathcal{M}_n\}) : \Gamma_n}$$

(4)
$$\frac{\Gamma,\ r \vdash \mathtt{method}({\phi_1}^{(k)}) : \Gamma_1 \cdots \Gamma_{n-1},\ r \vdash \mathtt{method}({\phi_n}^{(k)}) : \Gamma_n \quad \Gamma_n,\ r \vdash op_1 : \Gamma_1'\ \cdots\ \Gamma_{n-1}',\ r \vdash op_n : \Gamma_n'}{\Gamma,\ r \vdash \{\mathtt{method}({\phi_1}^{(k)}) \mathtt{=} op_1, \ldots, \mathtt{method}({\phi_n}^{(k)}) \mathtt{=} op_n\} : \Gamma_n'}$$

(5)
$$\frac{\Gamma \vdash \phi^{(k)} : \tau^{(k)}}{\Gamma,\ r \vdash \mathtt{method}(\phi^{(k)}) : \Gamma \cup \{r\ \mapsto\ \tau^{(k)}\}}$$

(6)
$$\frac{\Gamma,\ r \vdash op_1 : \Gamma' \quad \Gamma',\ r \vdash op_2 : \Gamma''}{\Gamma,\ r \vdash op_1\ \mathtt{then}\ op_2 : \Gamma''}$$

(7)
$$\frac{\Gamma \vdash \phi^{(k)} : {\tau_1}^{(k)} \quad \exists {\tau_2}^{(k)} \in \Gamma(r)\ \mid\ unify({\tau_1}^{(k)},\ {\tau_2}^{(k)})}{\Gamma,\ r \vdash \mathtt{call}(\phi^{(k)}) : \Gamma}$$

(8)
$$\frac{\Gamma(p) = \emptyset \quad \Gamma \vdash \phi^{(k)} : \tau^{(k)} \quad \Gamma \vdash \phi^{(l)} : \tau^{(l)} \quad \Gamma \vdash \phi^{(m)} : \tau^{(m)}}{\Gamma \vdash \phi^{(k)}\ \mathtt{=}\ p(\phi^{(l)})\ \mathtt{fault}\ \phi^{(m)} : \Gamma \cup \{p \mapsto (\tau^{(k)},\ \tau^{(l)},\ \tau^{(m)})\}}$$

(9)
$$\frac{\Gamma(p) = (\tau_{in}^{(k)},\ \tau_{out}^{(l)},\ \tau_{flt}^{(m)}) \quad \Gamma \vdash \phi^{(k)} : \tau^{(k)} \quad \Gamma \vdash \phi^{(l)} : \tau^{(l)} \quad \Gamma \vdash \phi^{(m)} : \tau^{(m)} \quad \vdash unify(\tau_{in}^{(k)},\ \tau^{(k)}) \quad \vdash unify(\tau_{out}^{(l)},\ \tau^{(l)}) \quad \vdash unify(\tau_{flt}^{(m)},\ \tau^{(m)})}{\Gamma \vdash \phi^{(k)}\ \mathtt{=}\ p(\phi^{(l)})\ \mathtt{fault}\ \phi^{(m)} : \Gamma}$$

(10)
$$\frac{\Gamma(\rho) = \emptyset \quad \Gamma \vdash \phi^{(k)} : \tau^{(k)} \quad \Gamma \vdash \phi_1 : atype \quad \Gamma \vdash \phi_2 : rtype}{\Gamma \vdash \rho(\phi^{(k)})\ \mathtt{=>}\ \mathtt{agent}(\phi_1,\ \phi_2) : \Gamma \cup \{\rho\ \mapsto\ \tau^{(k)}\}}$$

(11)
$$\frac{\Gamma \vdash \phi^{(k)} : {\tau_1}^{(k)} \quad \vdash unify(\Gamma(\rho),\ {\tau_1}^{(k)}) \quad \Gamma \vdash \phi_1 : atype \quad \Gamma \vdash \phi_2 : rtype}{\Gamma \vdash \rho(\phi^{(k)})\ \mathtt{=>}\ \mathtt{agent}(\phi_1,\ \phi_2) : \Gamma}$$

Figure 6: MAP typing rules.

the bottom left of each rule, in an approximately clockwise manner. For example, Rule 6 states that in order to check the sequence op_1 `then` op_2 in Γ and role r, we must first check op_1 in Γ, which yields Γ', and then we check op_2 in Γ' which yields the final environment Γ''. For brevity we have omitted the majority of the operations in MAP, as they are all defined similarly to Rule 6. We only consider the operations and actions which update the environment.

The typing process begins in Rule 3 which simply decomposes a scene into a collection of separate typing problems, where each role and associated set of methods are considered individually. In Rule 4 we consider a single role r, and all the methods associated with this role. The method declarations are checked (by Rule 5) before the operations which comprise the method, as we wish to have all the method declarations in the environment before typing the method calls. Method calls are typed in Rule 7, which ensures that a matching method can be found.

Procedure calls are typed by ensuring that every invocation of a procedure p is performed with the same argument types. This is accomplished by storing the argument types of the first call to the procedure in the environment, and then checking that all subsequent calls are consistent with the types in the environment. Rule 8 enters the argument types into the environment if no existing match is found, and Rule 9 checks subsequent calls for consistency.

The typing of message passing is performed in a similar way to the typing of procedures. We ensure that the types of the terms which are associated with a performative ρ are all consistent with one another. Rule 10 enters the performative types into the environment if no existing match is found, and Rule 9 checks subsequent performatives for consistency. The rules for message passing also ensure that ϕ_1 is a valid agent name, and ϕ_2 is a valid role name. We have only defined the typing rules for sending messages here. The rules for message receipt are essentially identical.

5 Conclusion

The purpose of this paper was to demonstrate that we can use protocols to assist in the construction, enactment, and verification of peer-to-peer (P2P) systems. Specifically, our previous research on agent protocols can readily be adapted to the composition of the associated web services. Our technique is founded on

MAP, which is a formally-defined and executable protocol language. MAP is a lightweight formalism, providing only a minimal set of operations. This was a deliberate choice as it allowed us to define the language and the type system without unnecessary complication. However, we are now considering many enhancements to the language that would make it more suited to P2P architectures. These enhancements include explicit support for different message communication patterns, improved fault-tolerance mechanisms, and additional data-types.

The MAP language can be used to encode a wide range of protocols, as previously demonstrated in our work on agent protocols. However, the hand-encoding of protocols into the MAP formalism remains a time-consuming process. We are therefore currently considering a number of approaches which will permit protocols to be constructed in a more efficient manner. The simplest approach is the provision of a graphical tool for constructing protocols. Beyond this, we would like to support the automatic generation of protocols. We have made some initial progress into the construction of protocols as an outcome of a planning process.

A further issue that we intend to address, concerns the discovery of web services. At present, the web services that we use to define a system must be known in advance. We would like to relax this restrictions, and allow a more flexible kind of composition, which allows for (semi-)automatic web service discovery and invocation. For this, we will need semantically annotated web services, on which we can reason about the behaviour of the services. This is currently an active area of research in the Semantic Web community.

References

[1] L. Cardelli. *Type Systems*, chapter 140, pages 2208–2236. The Computer Science and Engineering Handbook. CRC Press, 1997.

[2] M. Esteva, J. A. Rodríguez, C. Sierra, P. Garcia, and J. L. Arcos. On the Formal Specification of Electronic Institutions. In *Agent-mediated Electronic Commerce (The European AgentLink Perspective)*, volume 1991 of *Lecture Notes in Artificial Intelligence*, pages 126–147, 2001.

[3] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web Services Choreography Description Language (WS-CDL) Version 1.0. Available at `http://www.w3.org/TR/ws-cdl-10/`, December 2004.

[4] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[5] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (Part 1/2). *Information and Computation*, 100(1):1–77, September 1992.

[6] M. Oriol. Peer Services: From Description to Invocation. In *Proceedings of the First International Workshop on Agents and Peer-to-Peer Computing (AP2PC02)*, volume 2530 of *Lecture Notes in Computer Science*, pages 21–32, July 2002.

[7] M. Purvis, M. Nowostawski, S. Cranefield, and M. Oliveira. Multi-agent Interaction Technology for Peer-to-Peer Computing in Electronic Trading Environments. In *Proceedings of the Second International Workshop on Agents and Peer-to-Peer Computing (AP2PC03)*, volume 2872 of *Lecture Notes in Computer Science*, pages 150–161, July 2003.

[8] M. Singh. Peer-to-Peer Computing for Information Systems. In *Proceedings of the First International Workshop on Agents and Peer-to-Peer Computing (AP2PC02)*, volume 2530 of *Lecture Notes in Computer Science*, pages 15–20, July 2002.

[9] C. Walton. Model Checking Multi-Agent Web Services. In *Proceedings of the 2004 AAAI Spring Symposium on Semantic Web Services*, Stanford, California, March 2004. AAAI.

[10] C. Walton. Multi-Agent Dialogue Protocols. In *Proceedings of the Eighth International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January 2004.

[11] C. Walton and A. Barker. An Agent-based e-Science Experiment Builder. In *Proceedings of the 1st International Workshop on Semantic Intelligent Middleware for the Web and the Grid*, Valencia, Spain, August 2004.

[12] C. Walton and D. Robertson. Flexible Multi-Agent Protocols. In *Proceedings of UKMAS 2002. Also published as Informatics Technical Report EDI-INF-RR-0164, University of Edinburgh*, November 2002.