Protocols for Web Service Invocation

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

Abstract

The automatic invocation of a web service by an agent is a complex task which is currently being addressed by semantic markup techniques. However, it is difficult to define the computational aspects of a web service in this approach. In this paper we propose a protocol-based formalism which appears better suited to a representation of these issues. We define the syntax and semantics of a protocol language which express precisely how the interaction with a service should be performed, and how the service should be invoked. We also sketch an architecture for the execution of our protocols.

Introduction

A *web service* is a software system which utilises inter-operable mechanisms for distributed processing and machine-to-machine interaction on the World-Wide-Web. Web services are an increasingly important software technology. This is evident in the adoption of web services as the key programming paradigm for the Grid and the Semantic Web, and in the implementation of Multi-Agent Systems.

A web service is essentially a software component which resides at some fixed location on the web, typically inside a container on a web server. Interaction with web services is accomplished by remote procedure call (RPC) mechanisms. The advantage of web services over previous component-based middle-ware technologies (e.g. CORBA and DCOM) is that the protocols are web-oriented and lightweight. Web services are principally defined using two key standards: web service interfaces are specified in the Web Service Description Language (WSDL), and communication between web services is defined by the Simple Object Access Protocol (SOAP). In addition, there are many complementary standards which extend web services into different domains. For example, the Open Grid Services Architecture (OGSA-DAI) extends web services with facilities specific to the Grid, such as life-cycle management and Quality-of-Service guarantees. Similarly, the Business Process Execution Language for Web Services (BPEL4WS) extends web services with facilities to support business transactions.

In concert with the proliferation of web service technology, there has been an increasing focus of attention on the issues of *discovery* and *invocation* of web services. These issues are analogous to the search for, and retrieval of text-based web pages. Currently, the most popular solution is the use of a centralised *registry*, e.g. UDDI or ebXML. The registry stores the location of the WSDL specification, and a text-based categorisation of the service. However, the registry must be actively maintained by the owner of the web service, and the current systems are prone to stale and miscategorised entries.

To address the problems of the registry-based approach to discovery and invocation, there is significant research interest in the use of semantic web techniques in conjunction with web services. The semantic web aims to create a machine-readable web, allowing the integration of information from disparate sources to achieve the goals of end users. The intention is that this will be achieved through a process of *semantic markup* of information. The term *semantic web service* has recently been coined to denote a web service with such semantic markup. This markup is performed using formalisms such as OWL-S (OWL Services Coalition 2004), or WSMO (Roman, Keller, & Lausen 2004).

A semantic web service has associated ontological information which defines properties of the service. For example, OWL-S defines three main kinds of properties: the service profile; the service model; and the service grounding. Informally, these properties define the description of the service, the task performed by the service, and the concrete realisation of the service respectively. WSMO extends this list with non-functional properties which define features of the service, and goals which define the objectives of the service. The intention is that the semantic information will be used to automate the processes of web service discovery, invocation and composition. Nonetheless, significant challenges remain in the realisation of these goals.

In this paper, we focus on the challenges associated with the *automatic invocation* of a web service. This automation is necessary if the services are to be used by agents which are not under direct human control. We will assume that the appropriate web service has already been discovered, e.g. through a registry service. Our approach is based on the use of *interaction protocols* which define how to interact with a service. Our protocols are intended to be complementary to the use of both WSDL interface definitions and semantic markup techniques.

*This work is sponsored by the UK Engineering and Physical Sciences Research Council (Grant GR/N15764/01) Advanced Knowledge Technologies (AKT).

Related Work

The approach presented in this paper is derived from our earlier work on protocols for performing coordination in Multi-Agent Systems (Walton 2004b). This work was heavily influenced by the Electronic Institutions (Esteva *et al.* 2001) framework for expressing social norms in Multi-Agent Systems, and the π-calculus (Milner 1989) used in the formal specification of concurrent systems. The outcome of this work was the definition of an executable specification language of Multi-Agent Protocols, called MAP.

The provision of techniques for coordination and choreography between web services is currently a very active area of research. We have previously shown how MAP can be adapted to perform coordination between groups of web services (Walton & Barker 2004). In this work, our use of protocols for defining coordination is similar to that of (Singh *et al.* 2004). However, by basing our language on the π calculus, we do not restrict our attention to finite-state models. The π calculus has been previously used as the basis for the formalisation of web service choreography (Brogi *et al.* 2004), though the focus of this work is on the Web Service Choreography Interface (WSCI) proposal.

One of the key reasons for providing a formalisation of web service interactions it to allow verification to be performed. We have previously demonstrated the use of model checking techniques to ensure the correctness of our protocols (Walton 2004a). A similar approach has been demonstrated for web service flows in (Nakajima 2004). Consistency checking for web service interfaces has also been addressed in (Beyer, Chakrabarti, & Henzinger 2005).

In this paper, we take a different view on the coordination problem from before. In our previous work, we defined protocols which specified all of the interactions between a group of web services. However, we now focus our attention on a single web service, and define the interaction pattern for this service in isolation. The resulting protocol can then be used by an external agent to automatically invoke the service. These protocols are considerably simpler to execute and verify than those in the full MAP language. Automatic invocation of services has also been addressed by planning techniques (Pistore *et al.* 2004), and by logic-programming (McIlraith & Son 2002), though we are aiming for a more general solution that does not make specific requirements on the underlying technology.

The main alternative to the approach presented in this paper is the specification of data flows and parameter bindings which have recently been included in the OWL-S service model. These definitions are intended to express the flow of data from one process component to another. In principle, this should allow invocations patterns to be inferred. However, the current definitions are rather cumbersome in nature. In particular, the use of continuous data flows, makes it difficult to express dynamic behaviours or discrete events. By contrast, the definitions that we present in this paper are concise and lightweight, and can be readily validated. It is our opinion that these aspects of a web service are best defined separately from the service ontology, using a more appropriate formalism. On particular, we can express the *functional dependencies* between the operations which comprise the web service. Nonetheless, this does not preclude the use of out protocols in conjunction with a service ontology, as we describe later in the paper.

Interaction Protocols

At an intuitive level, the automated invocation of a web service by an external agent does not appear to be an overly complex task. However, there are a surprising number of issues that must be addressed to achieve the desired result. This can be demonstrated by considering the example of a job scheduler interface in Figure 1, which we will use throughout the paper. The name and argument types of each operation are specified in the interface, where `unit` denotes an empty argument list. The information contained in the interface is in effect the same as in the WSDL definition, though we do not use XML syntax here for readability.

```
initJob     : unit → id
setSchedule : id × sched → unit
getSchedule : id → sched
setScript   : id × script → unit
getScript   : id → script
getStatus   : id → status
commitJob   : id → unit
deleteJob   : id → unit
```

Figure 1: Job Scheduler Interface

In order to invoke the operations of the job scheduling service, we must consider (at least) the following issues. We need an intuition as to what each of the operations in the interface are for, and how they affect the state of the service. There is an implicit ordering of the operations, for example, `initJob` must be performed at the beginning of the interaction with the service, and `commitJob` at the end. We must also consider the different types which are used by the operations, for example, what a `sched` actually comprises. There are relationships between the arguments, for example, the `id` returned by `initJob` is used in the remaining operations. Finally, we need to understand how to obtain and construct appropriate data for each of the arguments.

Many of these issues can be addressed by the use of semantic markup. For example, we can specify the task performed by each operation, and the knowledge contained in the different arguments, through an appropriate ontology. However, the issues relating to computation, such the relationships among arguments, cannot readily be represented owing to the lack of variables in the underlying description logic. For example, in an OWL-S profile we can define the input of a service with `hasInput` declarations, but it is not a straightforward process to define the order of these inputs, or distinguish between different inputs of the same type. These issues suggest that a complementary approach would be desirable for the representation of the computational aspects of a service.

An ideal complement to semantic markup, which specifies the external behaviour of a service, would be the use of *algebraic specifications*. In this approach, the algebraic specification would define the internal behaviour of a ser-

vice in terms of first-order formulae. Reasoning could then be performed on the specification to infer the correct inputs to the service, the dependencies between operations, and the expected outcome. However, this is not the technique that we adopt in this paper. The primary reason for rejecting this approach is the difficulty of generating the specifications. In essence, the construction of a specification requires a reimplementation of a service in a first-order representation. This would be a difficult and error-prone task for non-trivial services.

The approach that we adopt in this paper has many of the advantages of algebraic specification, but does not require the laborious construction of first-order specifications. Instead, we take a *functional modelling* view which is a popular approach for the formal description of distributed systems. The basic idea is to characterise the input and output relationships of the system without specifying the internal behaviours. This is commonly called the *black-box* behaviour of a system. We define the input and output behaviour of the system using interaction protocols, which are easy to specify and understand. These protocols go far beyond the simple typing information which is present in a WSDL specification.

$p \in$ Protocol	::=	$\{\phi^n = \}\ op(\phi^k)$	(Invocation)
	|	p_1 `then` p_2	(Sequence)
	|	p_1 `or` p_2	(Choice)
	|	`do` p	(Repetition)
	|	ϵ	(Empty)
$\phi \in$ Term	::=	c	(Literal)
	|	v	(Variable)

Figure 2: Interaction Protocol Syntax

Our interaction protocols represent the ordering and dependencies between the operations of a web service. Figure 2 defines the abstract syntax for our protocol language, which is a simplified dialect of our MAP language for expressing web service coordination (Walton 2004a). The language describes patterns of invocations, expressed as a protocol p. There are three kinds of control flow in the language. The `then` operator expresses a sequential composition of p_1 followed by p_2. The `or` operator expresses a non-deterministic choice between p_1 and p_2. Finally, the `do` operator expresses zero or more iterations of p. The ϵ operator is a convenience which expresses an empty protocol step. The invocation of an operation op defines the input arguments ϕ^k and the (optional) output arguments ϕ^n. The arguments are terms ϕ, which are either literal constants c, or variable names v.

```
{job:id} = initJob() then
setSchedule(job:id, {mon, 11:00}:sched) then
setScript(job:id, {~/alarm.sh}:script) then
commitJob(job:id) then
{s:status} = getStatus(job:id)
```

Figure 3: Interaction Sequence

A protocol can express a single interaction with a web service. For example, Figure 3 illustrates one possible way to schedule a job with our interface. In this instance, we schedule the script `alarm.sh` to be executed on Monday at 11:00. The example makes clear the order of the operations, and the dependencies between variables.

The real power of the protocol language is in the specification of all possible interactions with a service, rather than a single interaction. Figure 4 defines a protocol for all possible interactions with our job scheduler. The protocol expresses the ordering of the operations and the relationship between the arguments. For example, the same variable name `s` is used in both `setSchedule` and `getSchedule`, indicating that they have the same value. The protocol resolves many ambiguities in the service. For example, `initJob` must be called before any other operations can be used, and the schedule and script cannot be set after the job is committed. The final step is a choice between `deleteJob` and the empty operator `e`, which makes the deletion optional.

```
{job:id} = initJob() then
setSchedule(job:id, s:sched) then
setScript(job:id, r:script) then
commitJob(job:id) then
do ( {s:sched} = getSchedule(job:id) or
     {r:script} = getScript(job:id) or
     {t:status} = getStatus(job:id) ) then
( deleteJob(job:id) or e )
```

Figure 4: Job Scheduler Protocol

It is important to note that the protocol gives us extra information that cannot simply be inferred by examining the interface types of the service. For example, there are five operations in our job scheduler which take a single `id` as input, but there is nothing to distinguish one from another. In principle, it would be possible to introduce extra types, to distinguish between different sequences. However, it is not possible to express iterations in this approach.

We can represent the control flow on a protocol as a non-deterministic finite-state system. Figure 5 illustrates the control flow in our example. This graph is a useful aid to understanding the structure of a protocol. However, it should be noted that the graph does not define the dependencies between the variables in the protocol. These variables give the system a memory capability and extend the power of the approach to the expression of infinite-state systems.

Protocol Execution

To illustrate how a protocol can assist in the automatic invocation of a web service, we will now outline an architecture for this purpose. The main components of the architecture are presented in Figure 6. We assume that each web service has an associated interface (e.g. WSDL) which defines the types of the operations, an ontology (e.g. OWL-S) which relates these types to concepts, and a protocol which defines how the service should be invoked. The web service is invoked by an agent which has its own local knowledge base, and can interact with an environment.

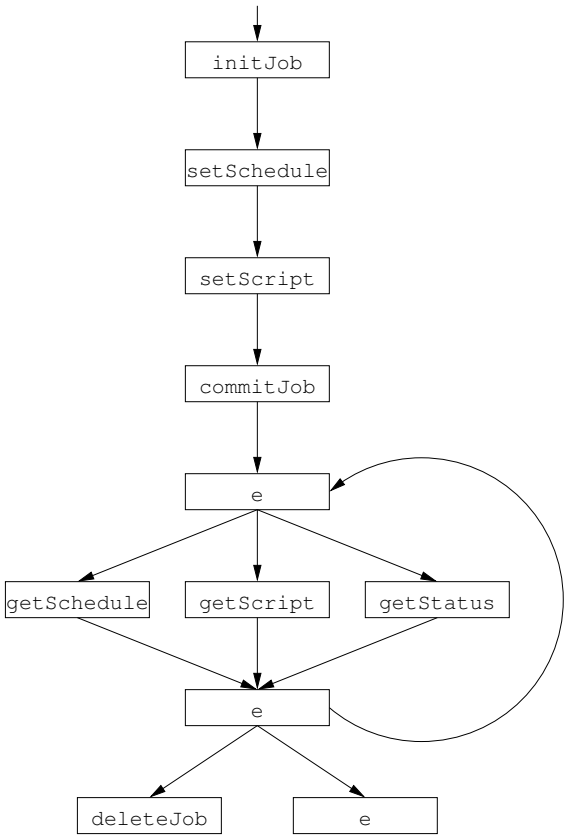

Figure 5: Protocol Graph

The automatic invocation process comprises six steps which are performed by the agent. The invocation process is driven by the protocol, which is retrieved in the first step. The agent follows the protocol until some parameters are required, at which point (step 2) a lookup is performed on the interface definition to obtain the types for the parameters. Subsequently, the ontology for the service is referenced to obtain the concepts relating to these types (step 3). At this point, the agent consults its own knowledge base to determine if it can provide appropriate values for the parameters (step 4). If the agent fails to obtain the necessary information from its own knowledge, the agent will consult the environment (e.g. other agents) to find an appropriate source of knowledge (step 5). If either step 4 or 5 is successful, then the agent can proceed to invoke the service (step 6). However, if appropriate values for the parameters cannot be found, the agent must backtrack through the protocol and attempt a different interaction. If no further interactions can be performed, then the protocol execution fails. The behaviour of the agent upon protocol failure is beyond the scope of our architecture. However, in an implementation, it is likely that the agent would refer back to a human user to supply the necessary knowledge.

We now discuss the execution of our protocols at a more formal level. The provision of a clean and unambiguous semantics was a primary consideration in the design of our

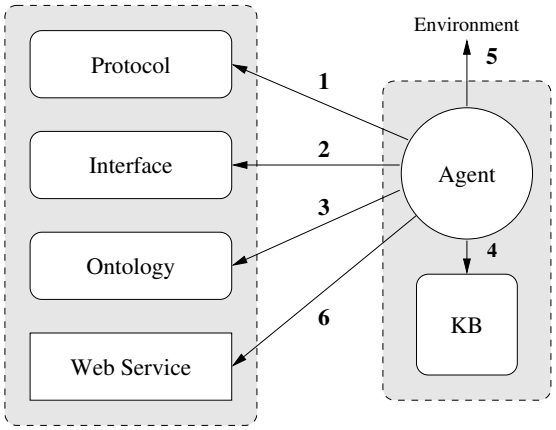

Figure 6: Execution Architecture

protocol language. The purpose of this semantics is to formally describe the meaning of the different language constructs, such that interaction protocols can be interpreted in a consistent manner. The semantics can also serve as a basis for the verification of the interaction protocols.

We have chosen to present this in a relational operational semantics formalism called *natural semantics*, so called because the evaluation of the relations is reminiscent of natural deduction. The natural semantics style is convenient because the entire evaluation can be captured within a (semi-)compositional derivation that can be reasoned about inductively. The rules of the semantics can be implemented directly (e.g. as Prolog Horn-clauses) and a derivation can be performed incrementally, in a depth-first manner, from the root to the leaves.

$$
\begin{array}{lll}
\Delta \in \text{Environment} & ::= & (PE,\ VE) \\
PE \in \text{Operations} & ::= & op \overset{map}{\mapsto} (\phi^k_{in},\ \phi^n_{out}) \\
VE \in \text{Variables} & ::= & v \overset{map}{\mapsto} \phi
\end{array}
$$

Figure 7: Evaluation Environment Δ

In natural semantics, we define relations between the initial and final *states* of *program fragments*. For our protocols, a program fragment is a part of a protocol p. The state is captured by an environment Δ, defined in Figure 7. The environment contains a definition of the operations in the service PE (i.e. the service interface), and the bound variables VE. For brevity we omit the rules for constructing the initial environment, and for checking well-formedness of the environment from our definition. However, we note that the initial environment will simply contain the input/output parameters taken directly from the WSDL interface.

We define the evaluation rules for the protocols in Figure 8. The form of these rules is $\Delta \vdash p \Rightarrow \Delta'$, where Δ is the state at the start of evaluation, p is the protocol fragment under consideration, and Δ' is the state on completion. We also define a substitution function, $\Delta \vdash subst(\phi) \Rightarrow \phi'$ which substitutes variables for terms, and the unification function $\Delta \vdash unify(\phi_1,\ \phi_2) \Rightarrow \Delta'$ which

$$\boxed{\Delta \vdash p \Rightarrow \Delta'}$$

$$\frac{}{\Delta \vdash \epsilon \Rightarrow \Delta}$$

$$\frac{\Delta \vdash p_1 \Rightarrow \Delta' \quad \Delta' \vdash p_2 \Rightarrow \Delta''}{\Delta \vdash p_1 \texttt{ then } p_2 \Rightarrow \Delta''}$$

$$\frac{\Delta \vdash p_1 \Rightarrow \Delta'}{\Delta \vdash p_1 \texttt{ or } p_2 \Rightarrow \Delta'}$$

$$\frac{\Delta \vdash p_2 \Rightarrow \Delta'}{\Delta \vdash p_1 \texttt{ or } p_2 \Rightarrow \Delta'}$$

$$\frac{\Delta \vdash p \Rightarrow \Delta_1 \ \cdots \ \Delta_{k-1} \vdash p \Rightarrow \Delta_k}{\Delta \vdash \texttt{do } p \Rightarrow \Delta_k}$$

$$\frac{\begin{array}{l}\Delta(op) = (\phi_{in}^k,\ \phi_{out}^n) \\ \Delta \vdash subst(\phi_2^k) \Rightarrow \phi_3^k \\ \Delta \vdash unify(\phi_{in}^k,\ \phi_3^k) \Rightarrow \Delta' \\ \Delta' \vdash eval(op) \Rightarrow \Delta'' \\ \{\ \Delta'' \vdash unify(\phi_{out}^n,\ \phi_1^n) \Rightarrow \Delta'''\ \}\end{array}}{\Delta \vdash \{\phi_1^n = \}\ op(\phi_2^k) \Rightarrow \Delta\{\cup\ \Delta'''\}}$$

$$\boxed{\Delta \vdash subst(\phi) \Rightarrow \phi'}$$

$$\begin{array}{l}\Delta \vdash subst(c) \Rightarrow c \\ \Delta \vdash subst(v) \Rightarrow \Delta(v) \\ \Delta \vdash subst(\phi^k) \Rightarrow (subst(\phi^1),\ \ldots,\ subst(\phi^k))\end{array}$$

$$\boxed{\Delta \vdash unify(\phi_1,\ \phi_2) \Rightarrow \Delta'}$$

$$\begin{array}{l}\Delta \vdash unify(c,\ c) \Rightarrow \Delta \\ \Delta \vdash unify(v,\ c) \Rightarrow \Delta[v\ \mapsto\ c] \\ \Delta \vdash unify(\phi_1^k,\ \phi_2^k) \Rightarrow \\ \quad unify(\phi_1^1,\ \phi_2^1) \cup \cdots \cup unify(\phi_1^k,\ \phi_2^k)\end{array}$$

Figure 8: Operational Semantics.

matches terms and binds matching variables. We note that the $\Delta \vdash eval(op) \Rightarrow \Delta'$ function evaluates the underlying operation *op* in the service.

The majority of the evaluation rules are straightforward. For example, there are two rules for the `or` operator, which express a non-deterministic choice between the first and second protocol fragments in the definition. However, the rule for invocation requires further explanation. A service invocation is defined syntactically as $\{\phi_1^n = \}\ op(\phi_2^k)$, which means that an operation *op* is invoked on the arguments ϕ_2^k, and optionally returns the arguments ϕ_1^n. This process is expressed semantically as follows. In the first step, we lookup the operation *op* in the environment Δ to obtain the input and output definitions ϕ_{in}^k and ϕ_{out}^n. We then perform a substitution on the input arguments which yields a new list ϕ_3^n. This substitution replaces any variables with their current values from the environment. We then unify the input parameters ϕ_{in}^k with the arguments list ϕ_3^n. The unification ensures that the arguments are a proper match for the input parameters, and binds any parameter variables to the arguments. The result of the unification is a new environment Δ' containing the newly bound variables. We then evaluate the operation *op* in the new environment, which yields a result environment Δ''. In we have specified any output parameters, then another unification step is performed. The end result is the initial environment composed with any output variables.

It should now be clear that the semantics define precisely the way in which a protocol should be interpreted by an agent. This is important as it allows a protocol to be implemented in a consistent manner. The semantics can also be used to check an interaction sequence for validity. We can simulate the execution of a protocol execution using these rules, without actually invoking any operations. If the process terminates, then we know that the sequence is valid.

Conclusion

In this paper we have presented a novel protocol based formalism which can represent the computational aspects of a web service, such as invocation order, and inter-argument dependencies. It should now be apparent that the protocol language is a powerful tool for expressing interactions with a web service. A protocol defines the interaction at a level of detail that goes beyond that currently provided by WSDL interface definitions, or semantic markup techniques. However, our approach is intended to be used in concert with service ontologies and interface definitions. Through the use of protocols, we have sufficient information to allow an agent to invoke the operations of a service in the correct order. Furthermore, given data of the correct type, in the knowledge-base of the agent, we can readily invoke the service in an automatic manner. Our current work is on the automatic generation of protocols using planning and machine learning techniques on the services, and the verification of protocols by automated means.

References

Beyer, D.; Chakrabarti, A.; and Henzinger, T. 2005. Web Service Interfaces. In *Proceedings of the 14th International World Wide Web Conference (WWW 2005)*, 148–159.

Brogi, A.; Canal, C.; Pimentel, E.; and Vallecillo, A. 2004. Formalizing Web Services Choreographies. In *Proceedings of the 1st International Workshop on Web Services and Formal Methods (WS-FM 2004)*.

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

McIlraith, S., and Son, T. 2002. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, 482–493.

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

Nakajima, S. 2004. Model-Checking of Safety and Security Aspects in Web Service Flows. In *Proceedings of the Fourth International Conference on Web Engineering (ICWE)*, 488–501.

OWL Services Coalition. 2004. OWL-S: Semantic Markup for Web Services (Version 1.1). Available at: `www.daml.org/services/`.

Pistore, M.; Barbon, F.; Bertoli, P.; Shaparau, D.; and Traverso, P. 2004. Planning and Monitoring Web Service Composition. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004)*.

Roman, D.; Keller, U.; and Lausen, H. 2004. Web Service Modelling Ontology (WSMO). Available at: `www.wsmo.org`.

Singh, M.; Chopra, A.; Desai, N.; and Mallya, A. 2004. Protocols for processes: programming in the large for open systems. *ACM SIGPLAN Notices* 39(12):73–83.

Walton, C., and Barker, A. 2004. An agent-based e-science experiment builder. In *Proceedings of the 1st International Workshop on Semantic Intelligent Middleware for the Web and the Grid*.

Walton, C. 2004a. Model Checking Multi-Agent Web Services. In *Proc. of the 2004 AAAI Spring Symposium on Semantic Web Services*.

Walton, C. 2004b. Multi-Agent Dialogue Protocols. In *Proceedings of the Eighth International Symposium on Artificial Intelligence and Mathematics*.