Model Checking Multi-Agent Web Services

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

Abstract

In this paper we address the verification of communication between agents participating in multi-agent web service systems. Our approach is founded on the application of model-checking techniques to protocols which express interactions between a group of agents in the form of a dialogue. We outline a web service architecture which supports the construction of multi-agent systems using web services technology. We then define a lightweight protocol language which can express a wide range of inter-agent dialogues, and we use the SPIN model checker to verify properties of this language. Our initial results show this approach has a satisfactory success rate in the detection of failures.

Introduction

A *web service* is a software system designed to support inter-operable machine-to-machine interaction over a network (Booth *et al.* 2003). At the time of writing, web services are a new and promising technology which standardises many aspects of distributed processing and communication on the World-Wide-Web. The appeal of web services over other inter-operability standards, such as CORBA, is the simplicity and flexibility of the architecture, which builds upon XML technologies. At the present time there are just two main standards which define the web services architecture: web services are specified in the Web Service Description Language (WSDL), and communication between web services is defined by the Simple Object Access Protocol (SOAP).

The web service architecture does not attempt to prescribe a specific programming technology or internal implementation of web services. Nonetheless, web services are closely related to the agent programming paradigm. The definition of the web services architecture (Booth *et al.* 2003) states:

"A web service is viewed as an abstract notion that must be implemented by a concrete agent. The agent is a concrete entity (a piece of software) that sends and receives messages, while the service is the set of functionality that is provided."

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

The architecture definition makes the assumption that the agents which provide the implementation of the web services are *software agents*, constructed as distributed software components, i.e. objects. This is a different kind of agent from the *intelligent agents* which are composed to form Multi-Agent Systems (MAS) (Franklin & Graesser 1997). However, this assumption does not preclude web services from being used to define MAS. We believe that web services are equally applicable to MAS and offer a promising alternative to the FIPA (FIP 1999) standard for specifying and constructing MAS.

The construction of MAS from web services introduces a different kind of web services architecture from the one presented in (Booth *et al.* 2003). In particular, the communication patterns between agents can be considerably more varied and complex. In a software agent architecture, inter-agent communication appears as *remote procedure calls* between distributed objects, and exists simply to enable distributed computation. Communication in this model consists primarily of the passing of values to remote procedures, and the receipt of messages containing the results. In an intelligent agent architecture, communication exists for the exchange of knowledge between the agents, and the communication can be viewed as a *dialogue* between individual autonomous agents. Intelligent agents may engage in complex scenarios such as auctions and negotiations. While both approaches involve the exchange of messages between agents, an intelligent agent architecture will clearly introduce an enhanced level of interaction between web services.

To define the interaction between intelligent agents, we require a suitable representation of the dialogue. The SOAP specification (Gudgin *et al.* 2003) defines a one-way stateless communication mechanism, but this is too low-level for our purposes. Although a dialogue can be viewed as a sequence of message exchanges between intelligent agents, the precise structure of the sequence is of particular importance. For example, in an auction dialogue the sequence of bids is an essential part of the bidding process. The pattern of message exchange, and the rules of the dialogue, can be represented by a *dialogue protocol*. This has previously been shown in a variety of formalisms, including Electronic Institutions (Esteva *et al.* 2001), Conversation Policy (Greaves, Holmback, & Bradshaw 1999), and Conversation Protocols (Fu, Bultan, & Su 2003). We have

also previously defined a Multi-Agent Protocol (MAP) language (Walton & Robertson 2002) for expressing dialogue protocols which we will outline in this paper. Our MAP language appears to be a good complement to SOAP, as both languages are independent of the message content or agent implementation.

Dialogue protocols specify complex, concurrent, and asynchronous patterns of communication between agents. Concurrency introduces *non-determinism* into the system 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. 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 chosen to perform the verification of our dialogue protocols using *model checking* (Clarke, Grumberg, & Peled 1999). This technique provides a degree of certainty that is unattainable through simulation and testing. Model checking is a formal method for automatically verifying properties of finite-state concurrent systems. A model checker normally performs an exhaustive search of the state space of a system to determine if a particular property holds. Given sufficient resources, the procedure will always terminate with a yes/no answer. This makes the model checking technique of significant practical value as a verification tool. Model checking has been used extensively in the verification of hardware systems, and there is a growing interest in applying this technique to software systems including BDI-based MAS, e.g. (Wooldridge *et al.* 2002). Model checking appears to be a good fit to the problems associated with communication between web services.

We use the SPIN (Holzmann 2003) model checker to perform a mechanical verification of our dialogue protocols. SPIN is a high-performance generic verification system which is designed for proving the correctness of process interactions. It accepts design specifications in its own language PROMELA (PROcess MEta-LAnguage), and verifies correctness claims specified as a Linear Temporal Logic (LTL) formula. SPIN has been in development for many years and includes a large number of techniques for improving the efficiency of the model checking, e.g. partial-order reduction, state-compression, and on-the-fly verification.

Our presentation in this paper is structured as follows: we describe how we use web services to construct MAS, and we define the Multi-Agent Protocol (MAP) language which we use to express our dialogue protocols. We then describe the essential features of a translation from MAP to PROMELAwhich enables us to perform model checking of our protocols. Lastly, we discuss our initial model checking results and outline an approach which permits a greater range of properties to be verified.

Multi-Agent Web Services

In order to construct a Multi-Agent System from a collection of web services, we must crucially define what is an *agent* in such a system. We have previously stated that MAS are constructed from intelligent agents, and web services are implemented as software agents. Therefore, it is necessary to unite these two notions of agency in order to produce the required definition. We can consider an intelligent agent as being an extension of a software agent with some intelligent behaviour, e.g. reasoning. Thus, the most apparent definition for an agent in our system would be an intelligent agent which implements some web service behaviour. We may construct these intelligent agents by equipping existing web services (implemented as software agents) with an agency stub containing the required intelligent behaviour. The resulting intelligent agents could then be composed into a MAS as shown in Figure 1.

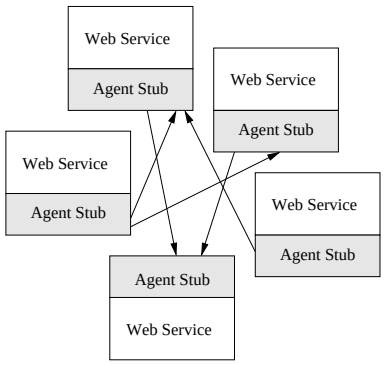

Figure 1: MAS Construction from Web Services

The construction of a MAS from web services which we have outlined above is certainly plausible. However, we do not adopt this approach as we believe there are some serious issues with this method. We will briefly outline these issues as they provide the motivation for our chosen technique.

The first issue is a result of the modification of the web services. We are in essence converting web services into intelligent agents and thereby breaking compatibility with existing web services and web service architectures. In order to deploy our modified services we require a new kind of architecture which permits intelligent behaviour, e.g. autonomy. Existing web service platforms, such as web containers, do not trivially permit this kind of behaviour and thus we would need to modify the existing frameworks or define a new agent platform.

The second issue concerns the agent stubs which provide the intelligent behaviour. In the general case, these stubs would provide a common set of functionality to the web services and be essentially identical in implementation. However, web services may be implemented in a variety of different languages and this would require the agent stub to be reimplemented in the language of the web service. Ensuring that the agent stubs implemented in different languages provide the same functionality and are completely compatible is a non-trivial task.

The final issue involves the spit between the agent stub and the web service. There is clearly a choice as to how much behaviour is generic and how much is specific to an individual agent. The most appropriate split is to utilise the web service as a library of decision procedures for the agents, and push the remaining behaviour into the agent stub. Web services are typically designed as a collection of remote procedures and are not particularly suited for the representation of state information. In this case, there is little need for the agent stub to be closely coupled to the web service.

Our chosen approach is illustrated in Figure 2. This approach does not break compatibility with the existing web service architecture as the only kinds of components we define are web services. The agents (labelled A) in the diagram are analogous to the agent stubs in the previous method. However, the agents are composed into a close-coupled MAS, and the MAS itself resides within a single web service. The web services external to the MAS provide the behaviours for the intelligent agents within the MAS. In effect we are implementing all of our intelligent agents within a single software agent.

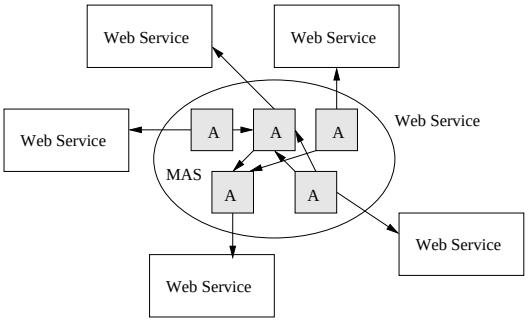

Figure 2: Alternative MAS Construction

Our alternative method addresses all of the issues which we have previously outlined and has some additional advantages. For example, the inter-agent communication now occurs within a single web service, rather than between distributed web services. External communication is only performed when an agent needs to make a decision. Another advantage is that we can reuse existing web services to provide agent decisions without the need to modify these services. Finally, we anticipate that we can readily compose our MAS web services to produce larger MAS, though this remains as future work.

The MAS web service in our approach is not a generic MAS platform, rather it is specific to a particular task as defined by a particular dialogue protocol. For example, a MAS web service may implement an auction task as defined by an auction dialogue. The MAS web service contains agents (A) which correspond to all of the participants in the dialogue. The external web services contain the decisions for the participants, e.g. bidding strategies.

We have implemented a platform for generating MAS web services from our MAP dialogue protocols. Figure 3 illustrates the main components of this platform. A protocol is defined in our MAP language, for example an auction

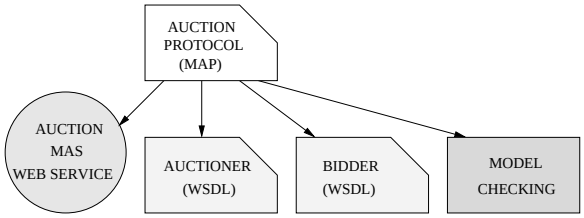

Figure 3: MAS Generation Platform

protocol. A web service, containing the MAS, is generated which implements the rules of the protocol and enforces the structure of the dialogue. The generation of the MAS web service is completely automatic in the platform. A WSDL specification is also generated for each of the roles in the dialogue, i.e. bidder and auctioneer. The WSDL document contains a specification of all the decision procedures for a particular agent role. It is necessary to implement these decisions manually as an external web service in order to participate in the MAS. An externally constructed web service, which implements a set of agent decisions, is registered with the MAS and this has the effect of generating an agent within the MAS. The MAS can be executed when an appropriate number of web services have been registered. We do not currently consider failures associated with web services in our platform.

The final component of the platform in Figure 3 is the model checking phase which is the subject of this paper. We do not discuss the other components of the platform in more detail here. The model checking phase detects potential problems in the execution of the MAS, e.g. synchronisation issues and deadlocks. These problems occur primarily as a result of unexpected interactions between agents. For example, the receipt of a stale bid may adversely affect an auction dialogue. In general, the prediction of undesirable behaviour in our dialogue protocols is a non trivial task. It should be noted that the model checking process is independent of the execution of the MAS, as the model checking is purely an operation on the protocol. Consequently, the results of this process are equally applicable to a distributed execution model as in Figure 1, or to the centralised model of Figure 2.

The MAP Language

The MAP language is a lightweight executable dialogue protocol language which provides a replacement for the statechart representation of protocols found in Electronic Institutions (Esteva *et al.* 2001). The concept of an agent *role* is central to our definition of a dialogue protocol. Agents participating in a protocol assume a fixed role for the duration of the protocol. For example, a negotiation protocol may involve agents with the roles of *buyer* and *seller*. The steps of the protocol which the agent follows in a dialogue will depend on the role of the agent. For example, an agent acting as a seller will typically attempt to maximise profit and will act accordingly in the negotiation. A role also identifies capabilities which the agent must provide. For example,

the buyer must have the capability to make buying decisions and to purchase items. Capabilities are related to the rational processes of the agent, as implemented by external web services, and are encapsulated by *decision procedures* in our definition.

P	::=	n`[`$\mathcal{A}$`]`	(Protocol)
A	::=	r`[`$\mathcal{M}$`]`	(Agent Role)
M	::=	`method(`$\phi^{(k)}$`) = `op	(Method)
op	::=	α	(Action)
	|	op_1 `then` op_2	(Sequence)
	|	op_1 `or` op_2	(Choice)
	|	op_1 `par` op_2	(Parallel)
	|	`waitfor` op_1 `timeout` op_2	(Iteration)
	|	`call(`$\phi^{(k)}$`)`	(Recursion)
α	::=	ϵ	(No Action)
	|	v `=` $p(\phi^{(k)})$	(Decision)
	|	M `=>` `agent(`$\phi^1,\ \phi^2$`)`	(Send)
	|	M `<=` `agent(`$\phi^1,\ \phi^2$`)`	(Receive)
M	::=	$\rho(\phi^{(k)})$	(Performative)
ϕ	::=	_ | a | r | c | v	(Terms)

Figure 4: MAP Abstract Syntax.

The abstract syntax of MAP is presented in Figure 4. We have also defined a corresponding concrete XML-based syntax for MAP which is used in our implementation. A protocol P is uniquely named n and defined as a set of agent roles $\mathcal{A}$, each of which define a set of methods $\mathcal{M}$. Agents have a fixed role r for the duration of the protocol, and are individually identified by unique names a. A method M can be considered a procedure where $\phi^{(k)}$ are the arguments. The initial protocol for an agent is specified by setting $\phi^{(k)}$ to be empty (i.e. $k = 0$). Protocols are constructed from operations op which control the flow of the protocol, and actions α which have side-effects and can fail. The interface between the protocol and the external web service, which defines the rational processes of the agent, is achieved through the invocation of decision procedures p. Interaction between agents is performed by the exchange of messages M which contain performatives ρ. Procedures and performatives are parameterised by terms ϕ, which are either variables v, agents a, roles r, constants c, or wild-cards _. Variables are bound to terms by unification which occurs in the invocation of procedures, the receipt of messages, or through recursive calls.

We will now define a simple auction protocol that will be used throughout the paper to illustrate the model checking process. Before we present the definition of this protocol in MAP, we consider a state-based description of the protocol, as shown in Figure 5. The state-based description is similar to a specification of the protocol in the Electronic Institutions framework. It is worth noting that MAP can also express protocols for which there is no state-based representation, e.g. protocols with parallel actions.

Our auction protocol is an attempt to simulate an English auction room. We do not impose any artificial constraints,

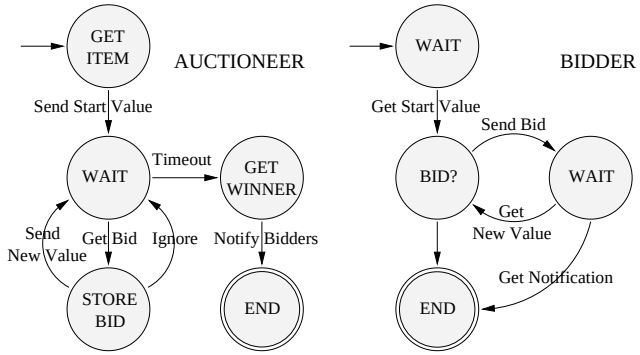

Figure 5: Auction Protocol States

such as turns or rounds, on the participants in the auction. The protocol assumes a single auctioneer agent and a variable number of bidder agents. The auction begins with the auctioneer sending out the starting value for a particular auction item. Each bidder then makes an internal decision whether to bid at the current value, and makes a bid if appropriate. When the auctioneer receives a valid bid, the bid value is incremented and the new value is sent to all of the bidders. The bidders then make a decision to bid at the new value. The auction continues until no further bids are received by the auctioneer and a timeout occurs (analogous to the "going, going, gone" ritual). At this point the winning bidder is notified and the auction concludes.

A definition of the auction protocol in MAP syntax is presented in Figure 6. For convenience, we distinguish between the different types of terms by prefixing variables names with `$`, and role names with `%`. We define two roles: `%auctioneer` and `%patient`. Each of these roles has two associated methods which define the protocol steps for the roles.

When exchanging messages through send and receive actions, a unification of terms in the definition `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, when the auctioneer informs the bidders of the starting value in line 5 of the protocol, the terms will match any agent whose role is a `%bidder`. Similarly, the receipt of the starting value in line 23 of the protocol will match any agent whose role is `%auctioneer`, and the name of this agent will be bound to the variable `$a`. We can therefore define broadcast and multi-cast communication modes.

The semantics of message passing corresponds to reliable, buffered, non-blocking communication. Sending a message will succeed immediately if an agent matches the definition, and the message M will be stored in a buffer on the recipient. Receiving a message involves an additional unification step. The message M supplied in the definition is treated as a template to be matched against any message in the buffer. For example, in line 10 of the protocol, a message must match `inform(bid, $bidval)`, and the variable `$bidval` will be bound to the second term in the message if the match is successful. Sending a message will fail if no

agent matches the supplied terms, and receiving a message will fail if no message matches the message template.

```
1 Auction_House[
2 %auctioneer{
3  method() =
4    $val = getValue() then
5    inform(start, $val) => agent(_, %bidder) then
6    call(bidloop, $val)
7
8  method(bidloop, $curr) =
9    waitfor
10    ((inform(bid, $bidval) <= agent($b, %bidder) then
11      $newval = recordBid($b, $bidval) then
12      inform(next, $newval, $b) => agent(_, %bidder)
13      then call(bidloop, $bidval))
14     or call(bidloop, $curr))
15   timeout
16     $win = getWinner() then
17     accept($win, $curr) => agent(_, %bidder)}
18
19 %bidder{
20 method() =
21  $id = getId() then
22  waitfor
23   (inform(start, $sval) <= agent($a, %auctioneer)
24    then $bidval = startBidding($sval, $id) then
25    inform(bid, $sval) => agent($a, %auctioneer)
26    then call(bidloop, $a, $sval))
27  timeout call()
28
29 method(bidloop, $a, $bidval) =
30  waitfor
31   ((inform(next, $newval, $high) <=
32            agent($a, %auctioneer) then
33     $highval = keepBidding($newval, $high) then
34     inform(bid, $newval) => agent($a, %auctioneer)
35     then call(bidloop, $a, $newval))
36    or accept($win, $wval) <= agent($a, %auctioneer))
37  timeout call(bidloop, $a, $bidval)}]
```

Figure 6: MAP Auction Protocol.

The send and receive actions complete immediately (i.e. non-blocking) and do not delay the agent. For this reason, all of the receive actions are wrapped by `waitfor` loops to avoid race conditions. For example, in line 22 the agent will loop until a message is received. If this loop was not present the agent may fail to find a starting value and the protocol would terminate prematurely. The advantage of non-blocking communication is that we can check for a number of different messages. For example, in lines 31 and 36 of the protocol, the agent waits for either a `next` message or an `accept` decision. A `waitfor` loop includes a `timeout` condition which is triggered after a certain interval has elapsed. This is used in lines 15 through 17 to determine the end of the auction.

At various points in the protocol, an agent is required to perform certain tasks, e.g. making a decision, or retrieving some information. This is achieved through the use of decision procedures. As stated earlier, a decision procedure provide an interface between the dialogue protocol and the rational processes of the agent, as implemented by external web services. In our language, a decision procedure p takes a number of terms as arguments and returns a single result in a variable v. For example, the `keepBidding` decision procedure in line 33 of the dialogue refers to an external decision procedure, which can be arbitrarily complex, e.g. based on a complex bidding strategy.

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. Our language also includes a `par` operator which evaluates op_1 and op_2 in parallel. This is useful when an agent is involved in more than one action simultaneously, though we do not use this in our example.

External data is represented by constants c in our language. We do not attempt to assign types to this data, rather we leave the interpretation of this data to the decision procedures. For example, in line 4 the starting value is returned by the `getValue` procedure, and interpreted by the `startBidding` procedure in line 24. Constants can therefore refer to complex data-types, e.g. currency, flat-file data, XML documents.

It should be clear that MAP is a powerful language for expressing multi-agent dialogues. We have used this language to specify a wide range of protocols, including a range of popular negotiation and auction protocols. It is important to note that MAP is not intended to be a general-purpose language, and therefore the relative paucity of features (e.g. no user-defined data-types) is entirely appropriate.

Model Checking MAP

The application of SPIN model checking to MAP protocols requires a representation of the protocols in PROMELA, which is the input language to the SPIN model checking process (Holzmann 2003). Of particular importance in this representation is the *level of abstraction* of the model on which the verification is performed. If the level of abstraction is too low, the state space will be too large and verification will be impossible. For example, it would be possible to construct a meta-interpreter for MAP protocols in PROMELA, but this would be unlikely to yield a sufficiently compact representation. Conversely, if the level of abstraction is too high then important issues will be obscured by the representation. Our chosen method of representation is a syntax-directed translation of the MAP protocols into PROMELA.

At an intuitive level there are a number of apparent similarities between MAP and PROMELA. For example, both are based on the notion of asynchronous sequential processes (or agents), and both assume that communication is performed via message passing. These high-level similarities significantly simplify the translation as we can translate MAP agents directly into PROMELA processes and agent communication into message passing over buffered channels. Nonetheless, the translation of the low-level details of MAP is not so straightforward as there are significant semantic differences in the execution behaviour of the languages.

There are essentially three points of semantic mismatch between MAP and PROMELA which we must address. The first of these concerns the order of execution of the statements in the language. In MAP, we assume a depth-first execution order, while PROMELA is based on guarded commands (Dijkstra 1975). The MAP language makes use of unification for the invocation of decision procedures, for recursion, and in message passing, while PROMELA has a call-by-value semantics. Finally, MAP assumes that messages can be retrieved in an arbitrary order (by unification), while PROMELA enforces a strict queue of messages. We will now sketch how these differences are handled in our translation.

We cannot readily represent the MAP execution tree in PROMELA as the language does not permit the definition of complex data structures. Thus, we initially considered an alternative formalism of the depth-first search using a stack-based algorithm. However, while in principle it is possible to encode a stack using PROMELA message buffers, it is our belief that this would yield an unnecessarily complex model. Our adopted solution involves flattening the execution tree through the translations shown in Figure 7. The templates shown are applied recursively, where $T(op)$ denotes a further translation of the operation op. We use a reserved variable `fail` to indicate whether a failure has occurred. This variable is tested on the execution of `then` and `or` operations. If a failure occurs, we skip all of the intermediate operations until an `or` node is encountered at which point the execution resumes. In this way we simulate the essential behaviour of the depth-first algorithm.

```
      MAP:   op1 then op2
PROMELA:     fail = false ;
             T(op1) ;
             if
               ::  (fail == false) -> T(op2)
               ::  else -> skip
             fi

      MAP:   op1 or op2
PROMELA:     fail = false ;
             T(op1) ;
             if
               ::  (fail == true) ->
                 fail = false ; T(op2)
               ::  else -> skip
             fi
```

Figure 7: Control Flow Translation.

Pattern matching is an essential part of the MAP language as it allows broadcast and multi-cast message passing to be succinctly expressed. For example, in our auction example, we send the starting value of the auction to all bidders in the operation `inform(start, $val) => agent(_, %bidder)`. Pattern matching is achieved through the unification of terms, which may bind variables to values. As PROMELA does not support pattern matching, we must perform a *match compilation* step in order to transform unification into a sequence of conditional tests. We do not describe the match compilation further here as there are many existing algorithms for performing this task.

The receipt of messages is a remaining difficulty in the translation process from MAP that we must address. We have previously stated that messages are stored in buffered channels in PROMELA, and we define a separate message buffer for each agent. However, a message buffer acts as a FIFO queue, and the messages must be retrieved in a strict order from the front of the queue. By contrast, messages in MAP are retrieved by unification and any message in the queue may be returned as a result. To simulate the behaviour required by MAP, we must remove all of the messages in the queue in turn and compare them with the required message by unification. The first message that is successfully matched is stored and the remaining messages are returned to the queue. It is worth noting that it is not enough simply to examine all the messages in the queue in-place, as we must remove a matching message, and this is only permitted from the front of the queue in PROMELA.

A pertinent issue in the translation process is the treatment of decision procedures in MAP protocols. The separation of rational processes from the communicative processes is a key feature in MAP. Nonetheless, the decision procedures are ultimately responsible for controlling the protocol and should be represented in some manner by our translation to PROMELA. To address this issue, we make the observation that the purpose of a decision procedure is to make a yes/no decision. Similarly, the purpose of the model checking process is to detect errors in the protocol and not in the decision procedures. Thus, we can in principle replace a decision procedure with any code that returns a yes/no decision. Furthermore, if this code returns a non-deterministic decision, the exhaustive nature of the model checking process will mean that all possible behaviours of the protocol will be explored. In other words, the model checker will explore all consequences for the protocol where the decision was yes, and all consequences where the decision was no.

```
1  /* Decision: keepBidding(newval, high) */
2  atomic {
3    if
4      :: true -> fail = true
5      :: true -> highval = PROC_KEEPBIDDING
6    fi }
```

Figure 8: `keepBidding` Decision Procedure.

Our translation of decision procedures into PROMELA is achieved by exploiting the non-determinism of guarded commands in the language. The semantics of guarded commands is such that if more than one guard is executable in a given situation, a non-deterministic choice is made between the guards. Therefore, the code fragment presented in Figure 8 can act as a suitable substitute for the `keepBidding` decision procedure from our auction protocol. The `true` guards in lines 4 and 5 respectively are both executable and a non-deterministic choice will be made between them. In the first case (line 4), we set the `fail` variable to indicate that a no decision was made. In the second case (line 5), corresponding to a yes decision, we do not need to take any

action. The decision is marked as `atomic` (line 2) as this improves the efficiency of the model checking.

The description of the translation which we have presented contains the essence of the translation from MAP into PROMELA. There are a number of residual issues such as the translation of parallel composition and the transmission of messages, but these are relatively straightforward extensions of the techniques that we have presented. The result of the translation is an specification of a protocol in PROMELA which replicates the semantics of the protocol as defined in MAP.

Results and Conclusions

In this paper we have presented a novel technique for constructing MAS using web services, and we have defined a language for representing Multi-Agent Dialogue Protocols (MAP). A formal semantics for the MAP language is presented in (Walton 2004). We have also outlined a syntax-directed translation from MAP into PROMELA as a means to perform verification with the SPIN model checker. Our translator has been applied to a number of protocols, including the auction example in this paper. As expected, the model checking process uncovered issues in these protocols which had remained hidden during simulation.

Our initial model checking experiments have focused on the *termination* of MAP protocols. This is an important consideration in the design of protocols, as we do not (normally) want to define protocols that cannot conclude. Non-termination can occur as a result of many different issues such as deadlocks, live-locks, infinite recursion, and message synchronisation errors. We also want to ensure that protocols do not simply terminate due to failure within the protocol. Therefore, we append the PROMELA code in Figure 9 to the end of each translated process. The test in line 2 will block if a failure has occurred, and the process will be prevented from reaching the end-state in line 3, i.e. the process will not terminate.

```
1  /* Check For Failure */
2  fail == false ;
3  end: skip
```

Figure 9: Check for Protocol Failure.

One of the key pragmatic issues associated with model checking is producing a state space that is sufficiently small to be checking with the available resources (1GB memory in our case). Hence, it is frequently necessary to make a number of simplifying assumptions in order to work within these limits. To achieve the model checking of our auction protocol, we were required to make two such simplifications.

The first simplification concerns the number of agents to use during checking. An ideal model would check the protocol for arbitrary numbers of agents (up to some finite bound). However, this would result in an unacceptable large model, and thus we are forced to fix the number of agents used in the checking process. We therefore fixed our protocol to a single auctioneer agent, and applied the checking algorithm with the number of bidders varying from 1 to 10. While it may be argued that the algorithm may contain errors which only become apparent with larger numbers of bidders, we believe that this is not the case. The errors in our protocols were rapidly exposed with just two bidders.

Our second simplification concerns the length of the auction process. The auction protocol which we have defined does not place any restriction on the length of the auction and is therefore in effect an infinite protocol (though in practise this will never be the case). Model checking is restricted to finite models, and therefore we must set a limit on the length of the auction. We therefore set a limit of 25 bids received by the auctioneer before the auction will be terminated. We believe this will cover all practical auction sizes.

The application of the SPIN model checker to the auction example, under the simplifications described above, uncovered two significant issues in the protocol which were previously undetected. The first of these issues concerns the recursive call in line 14 of the auction protocol. This redundant call occurs within a `waitfor` loop and has the effect of launching an additional auctioneer agent whenever a bid is not received. This call is redundant as it occurs within a loop and simply has the effect of restarting the loop. The effect is that a large number of unnecessary recursive calls are made, which results in a large number of processes being spawned in the PROMELA translation. The problem was not detected during simulation as the auction process always terminated within a small number of cycles. However, the problem was quickly discovered by the exhaustive model checking process. The result was an error message which stated that the number of processes had exceeded the capability of the checking algorithm. Removing the redundant call resulted in a model with an acceptable number of generated processes.

The second issue was uncovered as a direct result of the check for non-termination. Our auction protocol was designed under the assumption that certain decision procedures would never fail. We assumed that the `getValue()` procedure would always return a value to be used as the starting value of the auction, and that `getWinner()` would always return the winner (or a null value to indicate that there were no bids). However, our translation makes no such assumption as it substitutes a non-deterministic choice for each decision procedure. Therefore, the result is that if either the `getValue()` or `getWinner()` procedure fails, then the auctioneer agent will terminate with a failure, and the bidder agents will wait indefinitely.

The issue with decision procedures was resolved by introducing a new type of procedure into the MAP language, corresponding to a simple procedure that does not fail. We have found that it is often useful in the design of MAP protocols to have simple procedures which perform basic tasks, such as recording or returning values, and performing calculations. Amending the auction protocol with these simple procedures for the `getValue()` and `getWinner()` calls resulted in a model which successfully passed the model checking process.

The translation system which we have outlined is designed to perform *automatic* checking of MAP protocols. This makes the system suitable for use by non-experts who

```
/********** Auctioneer Decisions **********/
int highbidder, currentval;
#define START 10
#define INCREMENT 5

#define getValue() \
  atomic{ val = START ; currentval = START }

#define recordBid(bidder, bidval) \
 atomic { \
   if \
     :: bidval == currentval ->
          currentval = bidval + INCREMENT ; \
        newval = currentval; highbidder=bidder \
     :: else -> fail = true \
   fi }

#define getWinner() win = highbidder

/********** Bidder Decisions **********/
int myid, max;

#define startBidding(sval, id) \
  atomic { \
    if \
      :: sval > max -> fail = true \
      :: else -> myid = id; bidval = startval \
    fi }

#define keepBidding(newval, high) \
  atomic { \
    if :: high == myid -> fail = true \
       :: else -> if :: newval > max -> fail = true \
                     :: else -> newval = newval \
    fi            fi}
```

Figure 10: Minimal Procedure Implementation.

do not need to understand the model checking process. However, this approach restricts the kinds of properties of the protocols that we can check. In our auction example, we can check that the protocol terminates for a certain number of bidding rounds, but we cannot check that the highest bidder will win the auction. This is a result of our representation of decision procedures as abstract non-deterministic entities. In order to check a greater range of properties we must augment the PROMELA translation with additional information about the protocol. This information, and the resulting properties that we can check, are specific to the protocol. As an example, we can supply a minimal implementation of all the decision procedures in our auction protocol as shown in Figure 10. The decision procedures are implemented as macros which are manually inserted at the appropriate places in the translated code. This additional information captures the essence of an English auction protocol. We have used this code to verify that the highest bidder is always the winner of the auction.

We believe that the model checking results demonstrate a significant achievement in the design of reliable dialogue protocols, and consequently agent-based web services. Our current research is focused on improving the efficiency of the translation process to allow larger systems to be verified, and improving the reliability of the implementation by introducing a type inference mechanism.

References

Booth, D.; Haas, H.; McCabe, F.; Newcomer, E.; Champion, M.; Ferris, C.; and Orchard, D. 2003. *Web Services Architecture*. World-Wide-Web Consortium (W3C). Available at: `http://www.w3.org/TR/ws-arch/`.

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

Dijkstra, E. W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18(8):453–457.

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.

FIPA Foundation for Intelligent Physical Agents. 1999. *FIPA Specification Part 2 - Agent Communication Language*. Available at: `www.fipa.org`.

Franklin, S., and Graesser, A. 1997. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Proceedings of the Third International Workshop on Agent Theories*, number 1193 in Lecture Notes on Artificial Intelligence. Budapest, Hungary: Springer-Verlag. 21–36.

Fu, X.; Bultan, T.; and Su, J. 2003. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. In *Proceedings of the Eighth International Conference on Implementation and Application of Automata (CIAA'03)*, number 2759 in Lecture Notes in Computer Science, 188–200. Santa Barbara, California: Springer-Verlag.

Greaves, M.; Holmback, H.; and Bradshaw, J. 1999. What is a Conversation Policy? In *Proceedings of the Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents '99*.

Gudgin, M.; Hadley, M.; Mendelsohn, N.; Moreau, J. J.; and Nielsen, H. F. 2003. *SOAP Version 1.2 Specification*. World-Wide-Web Consortium (W3C). Available at: `http://www.w3.org/TR/soap12/`.

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

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

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

Wooldridge, M.; Fisher, M.; Huget, M. P.; and Parsons, S. 2002. Model Checking Multiagent systems with MABLE. In *Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*.