School of Informatics, University of Edinburgh

Centre for Intelligent Systems and their Applications

Model Checking Agent Dialogues in SPIN

by

Christopher Walton

Informatics Research Report EDI-INF-RR-0191

School of Informatics **January 2004**
http://www.informatics.ed.ac.uk/

Model Checking Agent Dialogues in SPIN

Christopher Walton

Informatics Research Report EDI-INF-RR-0191

SCHOOL *of* INFORMATICS
Centre for Intelligent Systems and their Applications

January 2004

Abstract :

The theory of Multi-Agent Systems (MAS) is typically concerned with a definition of the rational processes within the agents, or the communicative processes between agents. In this paper we are primarily interested in the latter, and in particular the problem of ensuring the correctness of communication. We address the problem by applying model checking techniques to protocols which express interactions between a group of agents in the form of a dialogue. We define a lightweight protocol language which can cleanly express a wide range of dialogues types, and we use the SPIN model checker to check the properties of this language.

Keywords : Multi-Agent Systems, Model Checking, Agent Dialogues

Model Checking Agent Dialogues in SPIN

Christopher D. Walton*

Centre for Intelligent Systems and their Applications (CISA),
School of Informatics, University of Edinburgh, UK.
Email: cdw@inf.ed.ac.uk Tel: +44-131-650-2718

Abstract. The theory of Multi-Agent Systems (MAS) is typically concerned with a definition of the rational processes within the agents, or the communicative processes between agents. In this paper we are primarily interested in the latter, and in particular the problem of ensuring the correctness of communication. We address the problem by applying model checking techniques to protocols which express interactions between a group of agents in the form of a dialogue. We define a lightweight protocol language which can cleanly express wide range of dialogues types, and we use the SPIN model checker to check the properties of this language.

1 Introduction

A popular theoretical basis for the specification of Multi-Agent Systems (MAS) is the Belief-Desire-Intention (BDI) model. This is derived the theory of *intentional reasoning*, developed by the philosopher Michael Bratman [Bra87], which introduced the notion that human behaviour can be predicted and explained through the use of attitudes (i.e. mental states). The BDI model has been enthusiastically adopted by the MAS community and underlies the popular Agent Communication Languages (ACLs) namely, the Knowledge Query and Manipulation Language (KQML) [PFPS+92] and the Foundation for Intelligent Physical Agents ACL (FIPA-ACL) [FIP99]. Nonetheless, there is a growing dissatisfaction with the BDI model as a basis for defining *inter-operable* agents between different agent platforms [Sin98].

Inter-operability in Multi-Agent Systems requires that agents built by different organisations, and using different software systems, are able to reliably communicate with one another in a common language with an agreed semantics. The problem with the BDI model as a basis for inter-operable agents is that although agents can be defined according to a commonly agreed semantics, it is not generally possible to verify that an agent is acting according to these semantics. This stems from the fact that it is not known how to assign mental states systematically to arbitrary programs. For example, we have no way of knowing whether an agent actually believes a particular fact. For the semantics

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

to be verifiable it would be necessary to have access to an agents' internal mental states which is not typically possible. This problem is known as the *semantic verification* problem and is detailed in [Woo00].

To understand why semantic verification is a highly-desirable property for an inter-operable agent system it is necessary to view the communication between agents as part of a coherent *dialogue* between the agents. According to the BDI model, the dialogue emerges from a sequence of communicative acts performed by an agent to satisfy their intentions. Furthermore, agents should be able to recognise and reason about the other agents intentions based upon these communicative acts. For example, according to the FIPA-ACL standard, the consequence of receiving an `inform` message is that the agent is entitled to believe that the sender believes the proposition in the message. There is an underlying *sincerity assumption* in this definition which demands that agents always act in accordance with their intentions. This assumption is considered too restrictive in an open environment as it will always be possible for an insincere agent to simulate any required internal state, and we cannot verify the sincerity of an agent as we have no access to is mental states.

In this paper we do not adopt a specific semantics of rational agency, or define a fixed model of interaction between agents. Our belief is that in a truly heterogeneous agent system we cannot constrain the agents to any particular model. For example, *web-services* [BHM+03] are rapidly becoming an attractive alternative to BDI-based MAS. Instead, we define a model of dialogue which separates the rational process and interactions from the actual dialogue itself. This is accomplished through the adoption of a *dialogue protocol* which exists at a layer between these processes. This approach has been adopted in the Conversation Policy [GHB99] and Electronic Institutions [ERS+01] formalisms among others. The definition presented in this paper differs in that dialogue protocol specifications can be directly executed. We define a lightweight language of Multi-Agent dialogue Protocols (MAP) as an alternative to the state-chart representation of protocols. Our formalism allows the definition of infinite-state dialogues and the mechanical processing of the resulting dialogue protocols. The underlying semantics of our language is derived from the field of process calculus. In particular MAP can be considered a heavily sugared variant of the Calculus of Communicating Systems (CCS) [Mil89].

Dialogue protocols specify complex concurrent and asynchronous patterns of communication between agents. This approach does not suffer from the semantic verification problem as the state of the dialogue is defined in the protocol itself, and it is straightforward to verify that an agent is acting in accordance with the protocol. Nonetheless, the design of concurrent protocols is a complex task, and this is no less a problem for protocols specified in MAP. Concurrency introduces *non-determinism* into a system which gives rise to a large number of potential problems, such as synchronisation, fairness, and deadlocks. It is difficult, even for an experienced protocol designer, to obtain a good intuition for the behaviour of a concurrent protocol, primarily due to the large number of possible interleavings which can occur. For example, the receipt of a stale or invalid bid may adversely

affect an auction dialogue. Therefore, the focus of this paper if on the verification of dialogue protocols specified in MAP.

Traditional debugging and simulation techniques cannot readily explore all of the possible behaviours of concurrent 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. The model checking technique has a particular appeal as it is an automated process, though it is limited to finite-state 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. Model checking has been applied with considerable success in the verification of concurrent hardware systems, and it is increasingly being used as a tool for verifying concurrent software systems, including multi-agent systems [BGS98,WFHP02].

One of the main issues in the verification of software systems using model checking techniques is the *state-space explosion* problem. The exhaustive nature of model checking means that the state space can rapidly grow beyond the available resources as the size of the model increases. This problem has affected previous attempts to model-check multi-agent systems e.g. [WFHP02], which use the BDI model as the basis for the verification process, limiting the applicability to very small agent models. This problem is mitigated to some extent by the adoption of MAP protocols in place of the BDI model. It is a fundamental concept of the BDI model that communicative acts are generated by agents in order to satisfy their intentions. Therefore, in order to model check BDI agents we must represent both rational and communicative processes in the model. By contrast, MAP protocols contain only a representation of the communicative processes of the agents and the resulting models are therefore significantly simpler.

We use the SPIN model checker [Hol03] to verify our MAP protocols, as we have no desire to construct our own model checking system. The SPIN model checker 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. SPIN accepts design specifications in its own language PROMELA (PROcess MEta-LAnguage), and verifies correctness claims specified as Linear Temporal Logic (LTL) formula. The verification is achieved by a translation from the MAP language to an abstract representation in PROMELA. We use this representation in SPIN to check a number of properties of the protocols, such as termination, liveness, and correctness. Our initial results have shown a good success rate in the detection of protocol errors.

Our presentation in this paper is structured as follows: in section 2 we define the MAP language which we use to express our dialogue protocols. In section 3 we present the essential features of a translation from MAP to PROMELA. Lastly, in section 4 we discuss our initial model checking results and outline an approach which permits a greater range of properties to be verified.

2 The MAP Language

The MAP language is a lightweight formalism for the expression of dialogue protocols. MAP was designed as a replacement for the state-chart representation of protocols found in Electronic Institutions [ERS+01]. We have redefined the core of the Electronic Institutions framework to provide an executable specification, while retaining the concepts of *institutions*, *scenes*, and *roles*.

The division of agent dialogues into *scenes* is a key concept in our protocol language. A scene can be thought of as a bounded space in which a group agents interact on a single task. The use of scenes divides a large protocol into manageable chunks. For example, a negotiation scene may be part of a larger marketplace institution. Scenes also add a measure of security to a protocol, in that agents which are not relevant to the task 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. Additional security measures can also be introduced into a scene, such as placing entry and exit conditions on the agents, though we do not deal with these here. However, we assume that a scene places barrier conditions on the agents, such that a scene cannot begin until all the agents are present, and the agents cannot leave the scene until the dialogue is complete.

The concept of an agent *role* is also central to our definition of a dialogue protocol. Agents entering a scene assume a fixed role which persists until the end of the scene. For example, a negotiation scene may involve agents with the roles of *buyer* and *seller*. The protocol which the agent follows in a dialogue will typically 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. These capabilities correspond to the rational processes of the agent and are encapsulated by *decision procedures* in our definition.

The abstract syntax of MAP is presented in Fig. 1. Agents are uniquely identified by a name a, and have a fixed role r for the duration of the scene. A scene n comprises an ordered sequence of protocols $P^{(k)}$. A protocol P can be considered a procedure where a, r, and $\phi^{(k)}$ 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 rational process 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,

$S \in$ Scene	$::= n\,\texttt{[}P^{(k)}\texttt{]}$	(Scene Definition)
$P \in$ Protocol	$::= \texttt{agent}(a,\ r,\ \phi^{(k)})\ \texttt{=}\ op$	(Agent Protocol)
$op \in$ Operation	$::= \alpha$	(Action)
	$\mid\ op_1\ \texttt{then}\ op_2$	(Sequence)
	$\mid\ op_1\ \texttt{or}\ op_2$	(Choice)
	$\mid\ op_1\ \texttt{par}\ op_2$	(Parallel Composition)
	$\mid\ \texttt{waitfor}\ op_1\ \texttt{timeout}\ op_2$	(Iteration)
	$\mid\ \texttt{agent}(\phi^{(k)})$	(Recursion)
$\alpha \in$ Action	$::= \epsilon$	(No Action)
	$\mid\ v\ \texttt{=}\ p(\phi^{(k)})$	(Decision Procedure)
	$\mid\ M\ \texttt{=>}\ \texttt{agent}(\phi^1,\ \phi^2)$	(Send)
	$\mid\ M\ \texttt{<=}\ \texttt{agent}(\phi^1,\ \phi^2)$	(Receive)
$M \in$ Message	$::= \rho(\phi^{(k)})$	(Performative)
$\phi \in$ Term	$::= v \mid a \mid r \mid c \mid \_$	(Protocol Terms)

Fig. 1. MAP Abstract Syntax.

as shown in Fig. 2. 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, 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 Fig. 3. For convenience, we distinguish between the different types of terms by prefixing variables names with `$`, role names with `%`, and agent names with `!`. We define two agents `!Auctioneer` and `!Bidder` which have roles `%auctioneer` and `%patient` respectively. We note that the protocol is compatible with multiple bidders, though these have been omitted for brevity.

When exchanging messages through send and receive actions, a unification of terms in the definition $\texttt{agent}(\phi^1,\ \phi^2)$ is performed, where ϕ^1 is matched

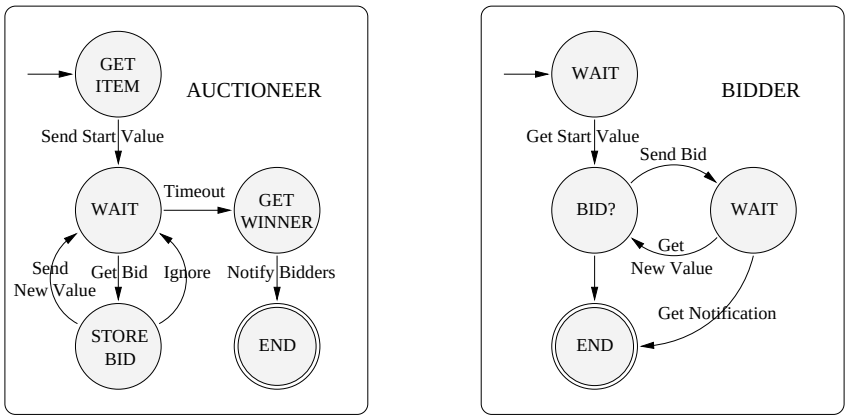

Fig. 2. Auction Protocol States

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 4 of the protocol, the terms will match any agent whose role is a `%bidder`. Similarly, the receipt of the starting value in line 20 of the protocol will match any agent whose role is `%auctioneer`, and the name of this agent will be bound to the variable `$auctioneer`. We can therefore define broadcast and multi-cast communications and our example will scale when more that two agents are present.

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 9 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.

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 19 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 28 and 33 of the protocol, the agent waits for either a `next` message or an `accept` decision. The `waitfor` loop includes a `timeout` condition which is triggered after a certain interval has elapsed. This is used in lines 14 through 16 to determine the end of the auction.

```
1 Auction_House[
2  agent(!Auctioneer, %auctioneer) =
3    $val = getValue() then
4    inform(start, $val) => agent(_, %bidder) then
5    agent(bidloop, $val)
6
7  agent(!Auctioneer, %auctioneer, bidloop, $currentval) =
8    waitfor
9     (inform(bid, $bidval) <= agent($bidder, %bidder) then
10      ($newval = recordBid($bidder, $bidval) then
11       inform(next, $newval, $bidder) => agent(_, %bidder) then
12       agent(bidloop, $bidval))
13     or agent(bidloop, $currentval))
14   timeout
15     ($winner = getWinner() then
16     accept($winner, $currentval) => agent(_, %bidder))
17
18 agent(!Bidder, %bidder) =
19   waitfor
20     (inform(start, $startval) <= agent($auctioneer, %auctioneer) then
21     $bidval = startBidding($startval, !Bidder) then
22     inform(bid, $startval) => agent($auctioneer, %auctioneer) then
23     agent(bidloop, $auctioneer, $startval))
24   timeout (agent())
25
26 agent(!Bidder, %bidder, bidloop, $auctioneer, $bidval) =
27   waitfor
28     (inform(next, $newval, $highbidder) <=
29                                  agent($auctioneer, %auctioneer) then
30       ($highval = keepBidding($newval, $highbidder) then
31       inform(bid, $newval) => agent($auctioneer, %auctioneer) then
32       agent(bidloop, $auctioneer, $newval)) or
33     accept($highbidder, $winval) <= agent($auctioneer, %auctioneer))
34  timeout(agent(bidloop, $auctioneer, $bidval))]
```

Fig. 3. MAP Auction Protocol.

At various points in the protocol, an agent is required to perform various 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. In our language, a decision procedure p takes a number of terms as arguments and returns a single result in a variable v. The actual implementation of the decision procedure is external to the dialogue protocol. For example, the `keepBidding` decision procedure in line 30 of the dialogue refers to an external

decision procedure, which can be arbitrarily complex, e.g. based on previous auction statistics, or according to a 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 3 the starting value is returned by the `getValue` procedure, and interpreted by the `startBidding` procedure in line 21. 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.

3 Model Checking MAP

The application of SPIN model checking to MAP protocols requires a representation of the protocols in PROMELA, which is the language used as input to the model checking process. 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-level, 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.

A syntax-directed translation is defined as a mapping from the abstract syntax of the source language to the abstract syntax of the target language. We translate from the MAP syntax, shown in Fig. 1, directly into the abstract representation of PROMELA shown in Fig. 4. Our PROMELA abstract syntax contains a representation the essential features of the full PROMELA language [Hol03]. For brevity we do not define the full translation process in this paper, rather we outline the key features of the translation.

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 per-

$Prog$	::= ($Def^{(k)}$, $Type^{(l)}$, $Dec^{(m)}$, $Proc^{(n)}$)	(Promela Program)
Def	::= `#define` $name$ $const$	(Macro Definition)
$Type$	::= `typedef` $tname$ `{` $Dec^{(k)}$ `}`	(User Defined Type)
Dec	::= $tname$ $Varef$	(Variable Declaration)
$Proc$	::= `init` $Stmt$	(Initial Process)
	| `proctype` $pname$ `(` $Expr^{(k)}$ `)` `{` $Stmt$ `}`	(Process Definition)
$Stmt$	::= $Expr$	(Expression)
	| Dec	(Local Declaration)
	| $Stmt$ `;` $Stmt$	(Sequence)
	| `skip`	(Skip)
	| `break`	(Escape)
	| `timeout`	(Timeout)
	| `end:` $Stmt$	(End Statement)
	| `atomic {` $Stmt$ `}`	(Atomic Statement)
	| `assert` $Expr$	(Assertion)
	| $Varef$ `:=` $Expr$	(Assignment)
	| `if` $Guard^{(k)}$	(Condition)
	| `do` $Guard^{(k)}$	(Iteration)
	| `run` $pname$ `(` $Expr^{(k)}$ `)`	(Process Generation)
$Guard$	::= `::` $Expr$ `->` $Stmt$ | `::` `else` `->` $Stmt$	(Guarded Statement)
$Expr$	::= $const$	(Constant)
	| $Varef$	(Variable)
	| $Expr_1$ `&&` $Expr_2$	(Conjunction)
	| ($Expr_1$ `->` $Expr_2$ `:` $Expr_3$)	(Conditional)
	| $Varef$ `!` $Expr$	(Send Message)
	| $Varef$ `?` $Expr$	(Receive Message)
	| op_u $Expr$ | $Expr_1$ op_b $Expr_2$	(Operations)
$Varef$	::= $vname$	(Variable Name)
	| $Expr$ `.` $vname$	(Structure Variable)
	| $Varef$ `[` $Expr$ `]`	(Array Reference)
	| $Varef$ `[` $Expr$ `]` `of {` $tname^{(k)}$ `}`	(Channel Reference)

Fig. 4. PROMELA Abstract Syntax.

formed 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 MAP, we assume a depth-first execution order, while PROMELA is based on guarded commands [Dij75]. 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 semantic differences are handled in our translation system.

A MAP protocol can be viewed as a tree structure, where the internal nodes of the tree are the `or` operations. The execution of the protocols proceeds incrementally in a depth-first manner through this tree, similar to proof search. Backtracking is performed when an operation fails, which occurs when message passing is unsuccessful, or a decision procedure returns a failure condition. At this point the execution resumes from the nearest `or` node, or fails completely if the root of the tree is reached. A precise semantics of the execution behaviour of MAP is presented in [Wal04].

MAP:	op_1 `then` op_2	op_1 `or` op_2

PROMELA:	`fail = false ;`
	$T(op_1)$ `;`
	`if`
	`:: (fail == false) ->`
	$T(op_2)$
	`:: else -> skip`
	`fi`
		`fail = false ;`
		$T(op_1)$ `;`
		`if`
		`:: (fail == true) ->`
		`fail = false ;` $T(op_2)$
		`:: else -> skip`
		`fi`

Fig. 5. Control Flow Translation.

We cannot readily represent the MAP execution tree in PROMELA as the language does not permit the definition of complex data structures. Consequently, 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 Fig. 5. 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.

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.

$$VE \vdash unify(\_,\ \phi) \Rightarrow VE$$
$$VE \vdash unify(\phi,\ \phi) \Rightarrow VE$$
$$VE \vdash unify(v,\ \phi) \Rightarrow VE[v\ \mapsto\ \phi]$$
$$VE \vdash unify({\phi_1}^{(k)},\ {\phi_2}^{(k)}) \Rightarrow unify(\phi_1^1,\ \phi_2^1) \cup \cdots \cup unify(\phi_1^k,\ \phi_2^k)$$

Fig. 6. Rules for Unification in MAP.

The rules for unification in MAP are straightforward and are shown in Fig. 6. We use VE to represent an environment which contains the values of the variables. The first rule states that any term matches a wild-card, and the second rule states that two terms match if they are identical. The third rule states that if the first term is a variable, the it will be bound to the second term in the environment VE. The final rule states that the unification rules should be applied to all of the terms in a sequence.

Before we define the match compilation, it is necessary to consider how we represent the different kinds of terms in PROMELA. There are five kinds of terms in MAP: variables, agents, roles, constants, and wild-cards. PROMELA has a very limited range of data-types, all of which are integer types, and thus it is necessary to translate the MAP terms into unique integers. We can take advantage of the fact that agents and roles in MAP are fixed throughout the evaluation of the protocol, and can therefore be statically mapped to integers. Constant names are also fixed and can be mapped in the same manner, and wild-cards are mapped to the integer 0 for convenience. For the variables we maintain an environment during the translation process which maps between variable names and integers. These integers are used to index an array, called `vars`, which is unique to each agent and contains values for all of its variables.

To improve the readability of the translation we define macros for all of the different terms. For example, the terms used in the auction example are represented by the macros in Fig. 7. The `VARS` macros define the size of the array which is required to hold all of the variables for a particular agent. The representation of messages in PROMELA is also shown in Fig. 7. We define a single `Message` structure which is used for all messages in the protocol. The structure

```
#define AGENT_BIDDER 1000        #define AGENT_AUCTIONEER 1001
#define ROLE_BIDDER 2000         #define ROLE_AUCTIONEER 2001
#define CONST_BID 3000           #define CONST_BIDLOOP 3001
#define VARS_AUCTIONEER 6        #define VARS_BIDDER 7
#define PERF_INFORM 4000         #define PERF_ACCEPT 4001
typedef Message
  { int agent ; int role ; int perf ; int arity ; int args[10] }
```

Fig. 7. Translation of MAP Terms and Messages.

contains the name and role of the agent sending the message, a performative, the arity of the message arguments, and an array of arguments.

To illustrate the match compilation process we present an example in Fig. 8, which shows the pattern matching associated with the receipt of a bid by the auctioneer. The translation demonstrates the representation of unification as a conjunction of equality tests. It is important to note that the variables are bound only after all the terms have been compared.

MAP:
```
inform(bid, $bidval) <= agent($bidder, %bidder) then ...
```

PROMELA:
```
if
  :: (mesg.role == ROLE_BIDDER && mesg.perf == PERF_INFORM
      && mesg.arity == 2 && mesg.args[0] == CONST_BID) ->
     vars[0] = mesg.agent ;
     vars[1] = mesg.args[1] ;
     ...
fi
```

Fig. 8. Match Compilation Example.

The example in Fig. 8 illustrates how we perform the unification of messages in PROMELA, but it does not show where these messages are obtained. The actual 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.

```
1  #define MBUFFER 50
2  chan messages = [MBUFFER] of {Message} ;
3  Message buff[MBUFFER] ;
4  int rx, ry ;
5
6  atomic {
7    rx = 0 ;
8    do
9      :: messages ? [buff[rx]] -> messages ? buff[rx] ; rx ++
10     :: else -> break
11   od;
12   ry = 0;
13   do
14     :: ry < rx ->
15       if
16         :: MATCH(buff[ry]) ->
17           ry ++ ;
18           do
19             :: ry < rx -> messages ! buff[ry] ; ry ++
10             :: else -> break
21           od;
22           break
23         :: else -> messages ! buff[ry] ; ry ++
24       fi
25     :: else -> fail = true ; break
26   od }
```

Fig. 9. Receipt of Messages.

A fragment of PROMELA code which performs a receive operation is shown in Fig. 9. The `messages` channel is the incoming message queue for the agent. The `buff` array is a temporary buffer which is used during the receive operation. The constant `MBUFFER` denotes the size of the message queue and temporary buffer, and the variables `rx` and `ry` are counters used in the algorithm. We denote the unification by `MATCH(buff[ry])` which corresponds to a match on the message in `buff[by]` as illustrated previously in Fig. 8. It is worth noting that we do not want any interference to the message queue while the operation is in progress, as this could corrupt the queue. Therefore, the entire receive operation is marked

as `atomic`. This also has the effect of simplifying the model checking operation by reducing the number of states in the resulting model.

The receive operation in PROMELA is implemented as follows. In lines 7 through 11 all of the queued `messages` are removed and placed in the buffer `buff`. The variable `rx` tracks the index of the buffer and contains the length of the buffer at the end of the copy operation. In lines 13 through 26 the messages in the buffer are examined in turn, and the variable `ry` tracks the position in the buffer. If a match is successful (line 16) then the variable `ry` is incremented (line 17) which has the effect of removing the message from the queue. We are only interested in the first match. Therefore, upon a successful match all of the remaining messages are copied back into the message queue (lines 18 though 21) and the loop is terminated (line 22). If the match is unsuccessful the message is simply returned to the message queue (line 23), and if no matches are found a failure condition is set (line 25).

A remaining issue in the translation process is the treatment of decision procedures in MAP protocols. Decision procedures are references to external rational processes which provide the reasoning capability to the agent system. For example, in our auction the bidder makes a decision to keep bidding: `$highval = keepBidding($newval, $highbidder)`. 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 the translation of decision procedures 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, based on these observations 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.

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 Fig. 10 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. In this case, we bind the name of the decision procedure to the result variable as this aids in the diagnosis of incorrect protocols. The decision is marked as `atomic` (line 2) as this improves the efficiency of the model checking.

```
1  /* Decision: keepBidding */
2  atomic {
3    if
4      :: true -> fail = true
5      :: true -> vars[5] = PROC_KEEPBIDDING
6    fi }
```

Fig. 10. Translation of `keepBidding` Decision Procedure.

The translation examples which we have presented contain 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.

4 Results and Conclusions

In this paper we have presented a novel language for representing Multi-Agent Dialogue Protocols (MAP), and we have outlined a syntax-directed translation from MAP into PROMELA for use in conjunction with the SPIN model checker. We have implemented the translation system as an extension of a Java toolkit, which we previously constructed to perform simulations of protocols specified in MAP [Wal04]. The translator has been applied to a number of protocols, including the auction example in this paper. We were pleased to find that the model checking process uncovered issues in these protocols which had remained hidden during simulation. We believe that this is a significant achievement in the design of reliable agent dialogue protocols.

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 scenes 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 Fig. 11 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.

One of the main 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

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

Fig. 11. Test for Protocol Failure.

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.

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 50 bids received by the auctioneer before the auction terminates with a winner.

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 13 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 rapidly triggered 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 pro-

cedure. 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.

```
1   /********** Auctioneer Decisions **********/
2   int highbidder;
3   #define START 10
4   #define INCREMENT 5
5   #define getValue() val = START
6   #define getWinner() winner = highbidder
7   #define recordBid(bidder, bidval) \
8     (bidval == currentval -> \
9       newval = bidval + INCREMENT ; highbidder = bidder : fail = true)
10
11  /********** Bidder Decisions **********/
12  int myid;
13  #define MAXBID 50
14  #define startBidding(startval, id) \
15    myid = id; (startval > MAXBID -> fail = true : bidval = startval)
16  #define keepBidding(newval, highbidder) \
17    (highbidder == myid -> skip : \
18      (newval > MAXBID -> fail = true : highval = newval))
```

Fig. 12. Implementation of Decision Procedures.

The translation system which we have outlined in this paper is designed to perform *automatic* checking of MAP protocols. This makes the system suitable for use by non-experts who do not need to understand the model checking process. However, this approach places restrictions on 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.

Our current research is aimed at extending the range of properties of dialogue protocols that can be checked with model checking. 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 Fig. 12. The decision procedures are implemented as macros which can be inserted at the appropriate place in translated code. This additional information captures the essence of an English auction protocol and should enable the checking of additional properties, though this remains as further work.

References

[BGS98] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model Checking Multiagent Systems. *Journal of Logic and Computation*, 8(3):401–423, June 1998.

[BHM+03] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. *Web Services Architecture*. World-Wide-Web Consortium (W3C), August 2003. Available at: `http://www.w3.org/TR/ws-arch/`.

[Bra87] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[ERS+01] 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)*, number 1991 in Lecture Notes in Artificial Intelligence, pages 126–147, 2001.

[FIP99] FIPA Foundation for Intelligent Physical Agents. *FIPA Specification Part 2 - Agent Communication Language*, April 1999. Available at: `www.fipa.org`.

[GHB99] M. Greaves, H. Holmback, and J. Bradshaw. What is a Conversation Policy? In *Proceedings of the Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents '99*, Seattle, Washington, May 1999.

[Hol03] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, September 2003.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[PFPS+92] R. Patil, R. F. Fikes, P. F. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The DARPA Knowledge Sharing Effort: Progress Report. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 777–788. Morgan Kaufmann, San Mateo, California, 1992.

[Sin98] M. P. Singh. Agent Communication Languages: Rethinking the Principles. *IEEE Computer*, pages 40–47, December 1998.

[Wal04] C. Walton. Multi-Agent Dialogue Protocols. In *Proceedings of the Eighth International Symposium on Artificial Intelligence and Mathematics (to appear)*, Fort Lauderdale, Florida, January 2004.

[WFHP02] M. Wooldridge, M. Fisher, M. P. Huget, and S. Parsons. Model Checking Multiagent systems with MABLE. In *Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*, Bologna, Italy, July 2002.

[Woo00] M. Wooldridge. Semantic issues in the verification of agent communication languages. *Autonomous Agents and Multi-Agent Systems*, 3(1):9–31, 2000.