Model Checking Agent Dialogues

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-(0)131-650-2718

Abstract. In this paper we address the challenges associated with the verification of correctness of communication between agents in Multi-Agent Systems. Our approach applies 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 express a wide range of dialogue types, and we use the SPIN model checker to verify properties of this language. Our early results show this approach has a high success rate in the detection of failures in agent dialogues.

1 Introduction

A popular basis for agent communication in Multi-Agent Systems (MAS) is the theory of *speech acts*, which is generally recognised to have come from the work of the philosopher John Austin [1]. This theory recognises that certain natural language utterances have the characteristics of physical actions in that they change the state of the world (e.g. declaring war). Austin identified a number of *performative verbs* which correspond to different types of speech acts, e.g. inform, promise, request. The theory of speech acts has been adapted for expressing interactions between agents by many MAS researchers, and this is most visible in the development of Agent Communication Languages (ACLs). The two most popular ACLs are currently the Knowledge Query and Manipulation Language (KQML) [21] and the Foundation for Intelligent Physical Agents ACL (FIPA-ACL) [12]. In these languages, the model of interaction between agents is based on the exchange of *messages*. KQML and FIPA-ACL define sets of performatives (message types) that express the intended meaning of the messages. These languages do not define the actual content of the messages and they assume a reliable method of message exchange.

In order to connect the theory of speech acts with the rational processes of agents, Cohen and Levesque defined a general theory of *rational action* [7]. This theory is itself based upon the theory of *intentional reasoning*, developed by the philosopher Michael Bratman [6], which introduced the notion that human behaviour can be predicted and explained through the use of attitudes (mental

* 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).

states), e.g. believing, fearing, hoping. In the general theory, speech acts are modelled as actions performed by agents to satisfy their intentions. The FIPA-ACL specification recognises this theory by providing a formal semantics for the performatives expressed in Belief-Desire-Intension (BDI) logic [22]. A BDI semantics for KQML has also been developed [17]. The combination of speech acts and intentional reasoning provides an appealing theoretical basis for the specification and verification of MAS [26]. Similarly, the KQML and FIPA standards provide useful frameworks for the implementation of MAS based upon these theories, e.g. JADE [2].

Nonetheless, there is a growing dissatisfaction with the mentalistic model of agency as a basis for defining *inter-operable* agents between different agent platforms [23, 16]. Inter-operability 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 to be verifiable it would be necessary to have access to an agents' internal mental states. This problem is known as the *semantic verification* problem and is detailed in [27].

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 theory of rational action, the dialogue emerges from a sequence of speech 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 speech acts. For example, according to the FIPA-ACL standard, if an agent receives an `inform` message then it 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 order to avoid the problems associated with the mentalistic model, and thereby express a greater range of dialogue types, a number of alternative semantics for expressing rational agency have been proposed. The two approaches that have received the most attention are a semantics based on social commitments, and a semantics based on dialogue games [18].

The key concept of the social commitment model is the establishment of shared commitments between agents. A social commitment between agents is a binding agreement from one agent to another. The commitment distinguishes between the creditor who commits to a course of action, and the debtor on whose behalf the action is done. Establishing a commitment constrains the subsequent actions of the agent until the commitment is discharged. Commitments are stored as part of the social state of the MAS and are verifiable. A theory which combines speech acts with social commitments is outlined in [11].

Dialogue games can trace their origins to the philosophical tradition of Aristotle. Dialogue games have been used to study fallacious reasoning, for natural language processing and generation, and to develop a game-theoretic semantics for various logics. These games can also be applied in MAS as the basis for interaction between autonomous agents. A group of agents participate in a dialogue game in which their utterances correspond to moves in this game. Different rules can be applied to the game, which correspond to different dialogue types, e.g. persuasion, negotiation, enquiry [25]. For example, a persuasion dialogue begins with an assertion and ends when the proponent withdraws the claim or the opponent concedes the claim. A framework which permits different kinds of dialogue games, and also meta-dialogues is outlined in [19].

There is an additional problem of verification of the BDI model, which we term the *concurrency verification* problem. A system constructed using the BDI model defines a complex concurrent system of communicating 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, 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.

In order to address the concurrency verification problem, a number of attempts have been made to apply model checking to models of BDI agents [3, 28, 5]. The model checking technique is appealing 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 and, given sufficient resources, the procedure will always terminate with a yes/no answer.

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. Thus, in order to successfully check a system it is necessary that the model is as small as possible. However, 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. This problem has affected previous attempts to model-check multi-agent systems e.g. [28], which use the BDI model as the basis for the verification process, limiting the applicability to very small agent models.

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* [4] 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 [13] and Electronic Institutions [10] 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 [14] representation of protocols. Our formalism allows the definition of infinite-state dialogues and the mechanical processing of the resulting dialogue protocols. MAP protocols contain only a representation of the communicative processes of the agents and the resulting models are therefore significantly simpler.

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, our experiences with defining dialogue protocols in MAP have shown that it is a difficult task to define correct protocols, even for simple dialogues. The problem is not related to the internal states of the agent, but rather as a result of unexpected interactions between agents. For example, the receipt of a stale bid may adversely affect an auction. In general, the prediction of undesirable behaviour in our dialogue protocols is non trivial. Thus, the focus of this paper if on the verification of dialogue protocols specified in MAP.

We use the SPIN model checker [15] 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 of our dialogue protocols 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 approach to translation is similar to [5], though we are primarily interested in checking general properties of inter-agent communication rather than specific BDI properties.

Our presentation in this paper is structured as follows: in Section 2 we define the syntax of the Multi-Agent Protocol (MAP) language. In Section 3 we specify the essential features of a translation from MAP to PROMELA which enables us to perform model checking of our protocols, and discuss our initial model checking results. We conclude in Section 4 with a discussion of future work.

2 The MAP Language

The MAP language is a lightweight dialogue protocol language which provides a replacement for the state-chart representation of protocols found in Electronic Institutions [10]. The underlying semantics of our language is derived from process calculus. In particular MAP can be considered a heavily-sugared variant of the Calculus of Communicating Systems (CCS) [20]. We have redefined the core of the Electronic Institutions framework to provide an executable specification, while retaining the concepts of *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.

$$\begin{array}{lcll}
P & ::= & n(r\texttt{\{}\mathcal{M}\texttt{\}})^{+} & \text{(Scene)} \\[4pt]
\mathcal{M} & ::= & \texttt{method}(\phi^{(k)})\ \texttt{=}\ op & \text{(Method)} \\[4pt]
op & ::= & \alpha & \text{(Action)} \\
 & | & op_1\ \texttt{then}\ op_2 & \text{(Sequence)} \\
 & | & op_1\ \texttt{or}\ op_2 & \text{(Choice)} \\
 & | & op_1\ \texttt{par}\ op_2 & \text{(Parallel)} \\
 & | & \texttt{waitfor}\ op_1\ \texttt{timeout}\ op_2 & \text{(Iteration)} \\
 & | & \texttt{call}(\phi^{(k)}) & \text{(Recursion)} \\[4pt]
\alpha & ::= & \epsilon & \text{(No Action)} \\
 & | & v\ \texttt{=}\ p(\phi^{(k)}) & \text{(Decision)} \\
 & | & M\ \texttt{=>}\ \texttt{agent}(\phi^1\texttt{,}\ \phi^2) & \text{(Send)} \\
 & | & M\ \texttt{<=}\ \texttt{agent}(\phi^1\texttt{,}\ \phi^2) & \text{(Receive)} \\[4pt]
M & ::= & \rho(\phi^{(k)}) & \text{(Performative)} \\[4pt]
\phi & ::= & \_\ \mid\ a\ \mid\ r\ \mid\ c\ \mid\ v & \text{(Terms)}
\end{array}$$

Fig. 1. MAP Abstract Syntax.

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. Capabilities are related to the rational processes of the agent and are encapsulated by *decision procedures* in our definition.

The abstract syntax of MAP is presented in Figure 1. We have also defined a corresponding concrete XML-based syntax for MAP which is used in our implementation. A scene protocol P is uniquely named n and defined as a (non-empty) sequence of roles r, each of which define a set of methods $\mathcal{M}$. Agents have a fixed role 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 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, agent names a, role names 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.

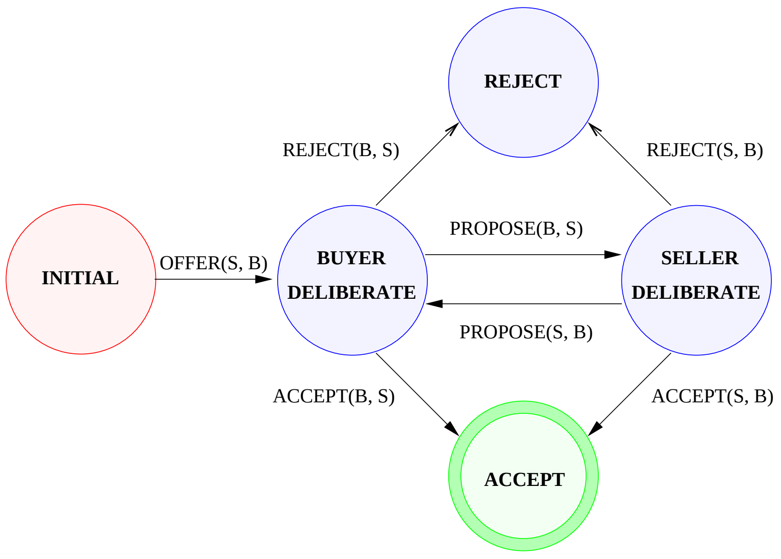

Fig. 2. Negotiation Protocol

We will now define a simple negotiation protocol, which will illustrate the MAP language and act as an example for model-checking. Before we present the definition of this protocol in MAP, we consider a state-based description of the protocol, as shown in Figure 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 finite-state representation, e.g. protocols with parallel actions.

Our negotiation protocol is an attempt to simulate a standard bargaining process between two parties (a buyer and a seller). We do not impose artificial constraints, such as turns or rounds, on the participants in the protocol. The negotiation begins with an offer from the seller to the buyer, which we denote with the message `OFFER(S, B)`. Upon receipt of the initial offer, the buyer enters a deliberative state, in which a decision is required. The buyer can accept or reject the offer in which case the protocol terminates. The buyer can also make a proposal to the seller `PROPOSE(B, S)`, e.g. an offer at a lower price. If a proposal is made to the seller, then the seller enters a deliberative state. The seller can in turn accept or reject the proposal, or make a counterproposal. If a counterproposal is made, the buyer deliberates further. Thus, the negotiation is effectively captured by a sequence of proposals and counter-proposals between the buyer and the seller.

A definition of the negotiation protocol in MAP syntax is presented in Figure 3. For convenience, we distinguish between the different types of terms by prefixing variables names with `$`, and role names with `%`. We define two roles: `%buyer` and `%seller`. Each of these roles has three associated methods which define the protocol states for the roles.

When exchanging messages through send and receive actions, a unification of terms in the definition `agent(`ϕ^1`, `ϕ^2`)` is performed, where ϕ^1 is matched against the agent name, and ϕ^2 against the agent role. For example, when the buyer receives the initial offer, in line 5 of the protocol, the terms will match any agent whose role is a `%seller`, and `$seller` will be bound to the name of the seller.

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 19 of the protocol, a message must match `accept($sellvalue)`, and the variable `$sellvalue` will be bound to the content of 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 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 18 the agent will loop until a message is received. If this loop was not present the agent may fail to find a response and the protocol would terminate prematurely. The advantage of

```
negotiate[
  %buyer{
    method() =
      waitfor
       (offer($value) <= agent($seller, %seller) then
        call(deliberate, $value, $seller))
      timeout (e)

    method(deliberate, $value, $seller) =
         ($newvalue = acceptOffer($value, $seller) then
          accept($value) => agent($seller, %seller))
      or ($newvalue = counterPropose($value, $seller) then
          propose($newvalue) => agent($seller, %seller) then
          call(wait, $newvalue))
      or  reject($value) => agent($seller, %seller)

    method(wait, $value) =
      waitfor
          (accept($sellvalue) <= agent($seller, %seller)
        or reject($oldvalue) <= agent($seller, %seller)
        or (propose($newvalue) <= agent($seller, %seller) then
            call(deliberate, $newvalue, $seller)))
      timeout (call(wait, $value))}

  %seller{
    method() =
      $value = getValue() then
      offer($value) => agent(_, %buyer) then
      call(wait, $value)

    method(wait, $value) =
      waitfor
          (accept($sellvalue) <= agent($buyer, %buyer)
        or reject($oldvalue) <= agent($buyer, %buyer)
        or (propose($newvalue) <= agent($buyer, %buyer) then
            call(deliberate, $newvalue, $buyer)))
      timeout (call(wait, $value))

    method(deliberate, $value, $buyer) =
         ($newvalue = acceptOffer($value, $buyer) then
          accept($value) => agent($buyer, %buyer))
      or ($newvalue = counterPropose($value, $buyer) then
          propose($newvalue) => agent($buyer, %buyer) then
          call(wait, $newvalue))
      or reject($value) => agent($buyer, %buyer)} ]
```

Fig. 3. MAP Negotiation Protocol.

non-blocking communication is that we can check for the receipt of a number of different messages. For example, in lines 19, 20, and 21 the protocol, the agent waits for either an `accept`, `reject`, or `propose` message respectively. The `waitfor` loop includes a `timeout` condition which is triggered after a certain interval has elapsed. The timeout is defined to restart the loop (in lines 23 and 37), though we could define an alternative behaviour, such as withdrawing from the negotiation. Timeouts give us a measure of fault tolerance in the presence of delays or failures.

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 `acceptOffer` decision procedure in line 31 of the dialogue refers to an external decision procedure, which can be arbitrarily complex, e.g. based on reputation, or according to some negotiation 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 27 the starting value is returned by the `getValue` procedure, and interpreted by the `acceptOffer` procedure in line 10. 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. It is important to note that MAP is only intended to express protocols, and is not intended to be a general-purpose language for computation. Therefore, the relative paucity of features, e.g. no user-defined data-types, is entirely appropriate. Furthermore, MAP is designed to be a lightweight protocol language and only a minimal set of operations has been provided. It is intended that MAP protocols will be automatically generated, e.g. from a planning system, or from visual tools such as ISLANDER [9].

A formal semantics for the MAP language has previously been presented in [24], together with an encoding of an auction protocol. We have used our language to specify a wide range of other protocols, including a range of popular negotiation and auction protocols. We have also restated the semantics of the FIPA-ACL performatives in MAP. Figure 4 gives a flavour of this transformation, with a (simplified) encoding of the FIPA `inform` performative.

FIPA Semantics: $< i,\ \texttt{inform}(j,\ \Phi) >$

$FP: \ B_i\Phi \wedge \neg B_i(Bif_j\Phi \vee Uif_j\Phi)$

$RE: \ B_j\Phi$

MAP Encoding:

```
method(inform, $p, $i, $j) =
  believe($i, $p) then
  not(believe($i, bif($j, $p)) then
  not(believe($i, uif($j, $p)) then
  inform(p) => agent($j, _) then
  assert(believe, $j, $p)
```

Fig. 4. Encoding of FIPA `inform` Performative.

3 Model Checking MAP

The first step in the application of SPIN model checking to MAP protocols is the construction of an appropriate system model. The underlying framework for modelling in SPIN is the Kripke structure, though this is well hidden underneath its own process meta-language PROMELA. SPIN translates the PROMELA language into Kripke structures, through a (loose) mapping of processes to states and channels to transitions. To generate the appropriate model for our MAP protocols, we perform a a translation from the MAP language to an abstract representation in PROMELA. Of particular importance in this translation 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.

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 MAP, we assume a depth-first execution order, while PROMELA is based on guarded commands [8]. 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. Furthermore, MAP assumes that messages can be retrieved in an arbitrary order (by unification), while PROMELA enforces a strict queue of messages. Finally, we must consider how to represent MAP decision procedures in our specification. We will now sketch how these semantic differences are handled in our translation system.

We cannot readily represent the MAP execution tree in PROMELA as the language does not permit the definition of complex data structures. Our adopted solution involves flattening the execution tree through the translations shown in Figure 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.

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

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

Fig. 5. Control Flow Translation.

Pattern matching is an essential part of the MAP language as it is used in method invocation, and in the exchange of messages. 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 unfold the 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.

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 required behaviour, 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. We note that it is not enough simply to examine all the messages in the queue in-place, as we must also remove a matching message.

A remaining issue in the translation process is the treatment of decision procedures, which are references to external rational processes. For example, in our negotiation the buyer may make a counterproposal, expressed in line 12: `$newvalue = counterPropose($value, %seller)`. 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 must 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, 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 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 Figure 6 can act as a suitable substitute for the `counterPropose` decision procedure. The decision is marked as `atomic` as this improves the efficiency of the model checking operation.

```
/* Decision: counterPropose */
atomic {
  if
    :: true -> fail = true
    :: true -> newvalue = PROC_COUNTERPROPOSE
  fi }
```

Fig. 6. Translation of `counterPropose` Decision Procedure.

We have now sketched the essence of the translation from MAP to PROMELA. There are a number of residual implementation issues, such as the implementation of parallel composition, but these can be readily represented in PROMELA. The result of the translation is an specification of a protocol in PROMELA which replicates the semantics of the protocol as defined in MAP.

Our initial model checking experiments with the SPIN model checker 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. The termination condition is the most straightforward to validate. Given that progress is a requirement in almost every concurrent system, the SPIN model checker automatically verifies this property by default. Every PROMELA process has one or more associated *end* states, which denote the valid termination points. The final state of a process is implicitly an end state. The termination condition states that every process eventually reaches a valid end state. This can be expressed as the following LTL formula, where `end1` is the end state for the first process, and `end2` is the end state for the second process, etc: $\Box(\Diamond(\texttt{end1} \wedge \texttt{end2} \wedge \texttt{end3} \wedge \cdots))$. We append the PROMELA code in Figure 7 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
```

Fig. 7. Test for Protocol Failure.

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. The negotiation protocol which we have defined does not place any restriction on the length of the deliberation process and is therefore in effect an infinite protocol. Model checking is restricted to finite models, and therefore we must set a limit on the length of the negotiation. We therefore set a limit of 50 cycles before the negotiation if forced to terminate.

An issue that was uncovered in the verification of the negotiation protocol is the treatment of certain decision procedures. Our protocol was designed under the assumption that the `getValue()` procedure would always return a value to be used as the starting value of the negotiation. However, our translation makes no such assumption as it substitutes a non-deterministic choice for each decision procedure. Therefore, the result is that if the `getValue()` procedure fails, then the seller agent will terminate with a failure, and the buyer will timeout. 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 negotiation protocol with a simple `getValue()` procedure resulted in a model which successfully passed the model checking process.

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. Our translator has been applied to a number of protocols, including the negotiation 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. In contrast with existing approaches to model checking MAS, our protocols remain acceptable in terms of memory and time consumption. Furthermore, we verify the actual protocol that will be executed, rather than an abstract version of the system.

Our MAP protocol language was designed to be independent of any particular model of rational agency. This makes the verification applicable to heterogeneous agent systems. Nonetheless, we recognise that the BDI model is still of significant importance to the agent community. To address this issue, we are currently defining a system which translates FIPA-ACL specifications into MAP protocols. We believe this will allow us to overcome the problems of the BDI model highlighted in the introduction, and will yield models that do not suffer from state-space explosion.

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 negotiation example, we can check that the protocol terminates, but we cannot check for a particular outcome. This is a result of our abstraction of decision procedures to 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 under verification. We have been able to verify protocol-specific properties with a hand-encoding of the decision procedures as PROMELA macros, but this relies on a detailed knowledge of the translation system. The provision of a general solution to the specification of protocol-specific properties remains as further work.

References

1. J. L. Austin. *How to Do Things With Words.* Oxford University Press, Oxford, UK, 1962.
2. F. Bellifemine, A. Poggi, and G. Rimassa. JADE: A FIPA-compliant agent framework. In *Proceedings of the 1999 Conference on Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'99)*, pages 97–108, London, UK, April 1999.

3. M. Benerecetti, F. Giunchiglia, and L. Serafini. Model Checking Multiagent Systems. *Journal of Logic and Computation*, 8(3):401–423, June 1998.
4. 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: `www.w3.org/TR/ws-arch/`.
5. R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model Checking AgentSpeak. In *Proceedings of the Second International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pages 409–416, Melbourne, Australia, July 2003. ACM.
6. M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.
7. P. R. Cohen and H. J. Levesque. Rational interaction as the basis for communication. *Intentions in Communication*, pages 221–256, 1990.
8. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
9. M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In *Proceedings of the First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pages 1045–1052, Bologna, Italy, July 2002. ACM press.
10. 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.
11. R. A. Flores and R. C. Kremer. Bringing Coherence to Agent Conversations. In *Proceedings of Agent-Oriented Software Engineering (AOSE 2001)*, volume 2222 of *Lecture Notes in Computer Science*, pages 50–67, Montreal, Canada, January 2002. Springer-Verlag.
12. Foundation for Intelligent Physical Agents. Fipa specification part 2 - agent communication language. Available at: `www.fipa.org`, April 1999.
13. 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.
14. D. Harel. Statecharts: A Visual Formalism for Computer System. *Science of Computer Programming*, 8(3):231–274, 1987.
15. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, September 2003.
16. Y. Labrou and T. Finin. Comments on the specification for FIPA '97 Agent Communication Language. Available at: `www.cs.umbc.edu/kqml/papers/`, February 1997.
17. Y. Labrou and T. Finin. Semantics and conversations for an agent communication language. In *Proceedings of the FIfteenth International Joint Conference of Artificial Intelligence (IJCAI-97)*, pages 584–591, Nagoya, Japan, August 1997.
18. N. Maudet and B. Chaib-draa. Commitment-based and Dialogue-game based Protocols–News Trends in Agent Communication Language. *The Knowledge Engineering Review*, 17(2):157–179, 2002.
19. P. McBurney and S. Parsons. Games that agents play: A formal framework for dialogues between autonomous agents. *Journal of Logic, Language and Information*, 11(3):315–334, 2002.
20. R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

21. 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.
22. A. S. Rao and M. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–344, 1998.
23. M. P. Singh. Agent Communication Languages: Rethinking the Principles. *IEEE Computer*, pages 40–47, December 1998.
24. C. Walton. Multi-Agent Dialogue Protocols. In *Proceedings of the Eighth International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January 2004.
25. D. N. Walton and E. C. W. Krabbe. *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning.* SUNY Press, 1995.
26. M. Wooldridge. *Reasoning about Rational Agents.* MIT Press, 2000.
27. M. Wooldridge. Semantic issues in the verification of agent communication languages. *Autonomous Agents and Multi-Agent Systems*, 3(1):9–31, 2000.
28. M. Wooldridge, M. Fisher, M. P. Huget, and S. Parsons. Model Checking Multi-agent systems with MABLE. In *Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*, Bologna, Italy, July 2002.