Model Checking for ACL Compliance Verification

Marc-Philippe Huget
Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK
mph@csc.liv.ac.uk

Michael Wooldridge
Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK
mjw@csc.liv.ac.uk

ABSTRACT

The problem of checking that agents correctly implement the semantics of an agent communication language has become increasingly important as agent technology makes its transition from the research laboratory to field-tested applications. In this paper, we show how model checking techniques can be applied to this problem. Model checking is a technique developed within the formal methods community for automatically verifying that finite-state concurrent systems implement temporal logic specifications. We first describe a variation of the MABLE multiagent BDI programming language, which permits the semantics (pre- and post-conditions) of ACL performatives to be defined separately from a system where these semantics are used. We then show how assertions defining compliance to the semantics of an ACL can be captured as claims about MABLE agents, expressed using MABLE's associated assertion language. In this way, compliance to ACL semantics reduces to a conventional model checking problem. We illustrate our approach with a number of short case studies.

Keywords

Agents, Model Checking, Agent Communication Language, Semantics, Verification

1. INTRODUCTION

The problem of checking that agents correctly implement the semantics of an agent communication language has become increasingly important as agent technology makes its transition from the research laboratory to field-tested applications. In this paper, we show how *model checking techniques* can be applied this problem, by making use of our MABLE language for the automatic verification of multiagent systems [20].

Model checking is a technique that was developed within the formal methods community for automatically verifying that finite-state systems implement temporal logic specifications [1]. The name "model checking" arises from the fact that verification can be viewed as a process of checking that the system is a model that validates the specification. The principle underpinning the approach is that the possible computations of given a system $\mathcal{S}$ can be understood as a directed graph, in which the nodes of the graph correspond to possible states of $\mathcal{S}$, and arcs in the graph correspond to state transitions. Such directed graphs are essentially *Kripke structures* — the models used to give a semantics to temporal logics. Crudely, the model checking verification process can then be understood as follows: Given a system $\mathcal{S}$, which we wish to verify satisfies some property φ expressed in a temporal logic L, generate the Kripke structure $M_{\mathcal{S}}$ corresponding to $\mathcal{S}$, and then check whether $M_{\mathcal{S}} \models_L \varphi$, i.e., whether φ is L-valid in the Kripke structure $M_{\mathcal{S}}$. If the answer is "yes", then the system satisfies the specification; otherwise it does not.

Our approach to model checking for the ACL compliance problem is as follows. In a previous paper [20], we described our first implementation of the MABLE language. MABLE is a language intended for the design and automatic verification of multi-agent systems; it is essentially an imperative programming language, augmented by some features from Shoham's agent-oriented programming paradigm [15]: in particular, agents in MABLE possess data structures corresponding to their *beliefs*, *desires*, and *intentions* [19]. A MABLE system may be augmented by a number of *claims* about the system, expressed in a simplified form of the $\mathcal{LORA}$ language given in [19]. Our MABLE compiler translates the MABLE system into processes in PROMELA, the input language for the SPIN model checking system [8, 9]; claims are translated into SPIN-format LTL formulae. The SPIN system can then be directly invoked to determine whether or not the original system satisfied the original claim.

In this paper, we show how MABLE has been extended in two ways to support ACL compliance testing. First, we have added a feature to allow programmers to define the semantics of ACL performatives separately from a program that makes use of these performatives, thus making it possible for the same program to exhibit different behaviours with different semantics. Second, we have extended the MABLE claims language to support a dynamic logic-style "`happens`" construct: we can thus write a claim that expresses that (for example) whenever an agent performs action α, property φ eventually follows. By combining these two features, we can automatically verify whether or not an agent respects ACL semantics.

The remainder of the paper is structured as follows. We begin with an overview of the ACL compliance checking problem. We then describe a variation of the MABLE multiagent BDI programming language, which permits the semantics (pre- and post-conditions) of ACL performatives to be defined separately from a system where these semantics are used. We then show how assertions defining compliance to the semantics of an ACL can be captured as claims about MABLE agents, expressed using MABLE's associated assertion language. In this way, compliance to ACL semantics reduces to a conventional model checking problem. We illustrate our approach with a number of short case studies, and conclude with a discussion of future work.

2. ACL COMPLIANCE VERIFICATION

One of the main reasons why multi-agent systems are currently a major area of research and development activity is that they are seen as a key enabling technology for the Internet-wide electronic commerce systems that are widely predicted to emerge in the near future [6]. If this vision of large-scale, open multi-agent systems is to be realised, then the fundamental problem of *inter-operability* must be addressed. It must be possible for agents built by different organisations using different hardware and software platforms to safely communicate with one-another via a common language with a universally agreed semantics. The inter-operability requirement has led to the development of several standardised *agent communication languages* (ACLs) [11, 5]. The development of these languages has had significant input from industry, and particularly from European telecommunications companies.

However, in order to gain acceptance, particularly for sensitive applications such as electronic commerce, it must be possible to determine whether or not any system that claims to *conform* to an ACL standard actually does so. We say that an ACL standard is *verifiable* if it enjoys this property. FIPA — currently the main standardisation body for agent communication languages — recognise that "demonstrating in an unambiguous way that a given agent implementation is correct with respect to [the semantics] is not a problem which has been solved" [5], and identify it as an area of future work. (Checking that an implementation respects the *syntax* of an ACL such as that proposed by FIPA is, of course, trivial.) If an agent communication language such as FIPA's is ever to be widely used — particularly for such sensitive applications as electronic commerce — then such compliance testing is important. However, the problem of compliance testing (*verification*) is not actually given a concrete definition by FIPA, and no indication is given of how it might be done.

In [18], the verification problem for agent communication languages was formally defined for the first time. It was shown that verifying compliance to some agent communication language reduced to a verification problem in exactly the sense that the term in used in theoretical computer science. To see what is meant by this, consider the semantics of FIPA's `inform` performative [5, p25]:

$$\begin{array}{ll} \langle i, \mathit{inform}(j, \varphi)\rangle & \\ \quad \text{FP:} & B_i\varphi \wedge \neg B_i(\mathit{Bif}_j\varphi \vee U_j\varphi) \\ \quad \text{RE:} & B_j\varphi \end{array} \qquad (1)$$

Here $\langle i, \mathit{inform}(j, \varphi)\rangle$ is a FIPA message: the message type (performative) is `inform`, the content of the message is φ, and the message is being sent from i to j. The intuition is that agent i is attempting to convince (inform) agent j of the truth of φ. The FP and RE define the semantics of the message: FP is the *feasibility pre-condition*, which states the conditions that must hold in order for the sender of the message to be considered as sincere; RE is the *rational effect* of the message, which defines what a sender of the message is attempting to achieve. The B_i is a modal logic connective for referring to the beliefs of agents (see e.g., [7]); *Bif* is a modal logic connective that allows us to express whether an agent has a definite opinion one way or the other about the truth or falsity of its parameter; and U is a modal connective that allows us to represent the fact that an agent is "uncertain" about its parameter. Thus an agent i sending an *inform* message with content φ to agent j will be respecting the semantics of the FIPA ACL if it believes φ, and it it not the case that it believes of j either that j believes whether φ is true or false, or that j is uncertain of the truth or falsity of φ.

It was noted in [18] that the FP acts in effect as a *specification* or *contract* that the sender of the message must satisfy if it is to be considered as respecting the semantics of the message: an agent respects the semantics of the ACL if, when it sends the message, it satisfies the specification. Although this idea has been understood in principle for some time, no serious attempts have been made until now to adopt this idea for ACL compliance testing.

We note that a number of other approaches to ACL compliance testing have been proposed in the literature. Although it is not the purpose of this paper to contribute to this debate, we mention some of the key alternatives. Pitt and Mamdani defined a *protocol-based semantics* for ACLs [12]: the idea is that the semantics of an ACL are defined in terms of the way that they may be used in the context of larger structures, i.e., protocols. Singh championed the idea of *social* semantics: the idea that an ACL semantics should be understood in terms of the observable, verifiable changes in social state (the relationships between agents) that using a performative causes [16].

3. MABLE

MABLE is a language intended for the design and automatic verification of multi-agent systems. The language was introduced in [20]; here, we give a high-level summary of the language, and focus in detail on features new to the language since [20].

Agents in MABLE are programmed using what is essentially a conventional imperative programming language, enriched with some features from agent-oriented programming languages such as AGENT0 [15], GOLOG [10], and AGENTSPEAK [13]. Thus, although the control structures (iteration, sequence, and selection) resemble (and indeed are closely modelled on) those found in languages such as C, agents in MABLE have a *mental state*, consisting of data structures that represent the agent's beliefs, desires, and intentions (cf. [19]). The semantics of MABLE program constructs are defined with respect to the mental states of the agents that perform these statements. For example, when an agent executes an assignment operation such as

```
x = 5
```

then we can characterise the semantics of this operation by saying that it causes the agent executing the instruction to subsequently believe that the value of `x` is 5.

In addition, MABLE systems may be augmented by the addition of formal *claims* made about the system. Claims are expressed using a (simplified) version of the belief-desire-intention logic $\mathcal{LORA}$ [19], known as $\mathcal{MORA}$ [20]; we describe this language in more detail below.

The MABLE language has been fully implemented. The implementation makes use of the SPIN system [8, 9], a freely available model-checking system for finite state systems. Developed at AT&T Bell Labs, SPIN has been used to formally verify the correctness of a wide range of finite state distributed and concurrent systems, from protocols for train signalling to autonomous spacecraft control systems [9]. SPIN allows claims about a system to be expressed in propositional Linear Temporal Logic (LTL): SPIN is capable of automatically checking whether or not such claims are true or false.

The MABLE compiler takes as input a MABLE system and associated claims (in $\mathcal{MORA}$) about this system (see Figure 1). MABLE generates as output a description of the MABLE system in PROMELA, the system description language for finite-state systems used by the SPIN model checker, and a translation of the claims into the LTL form used by SPIN for model checking. SPIN can then be used to automatically verify the truth (or otherwise) of the claims, and simulate the execution of the MABLE system, using the PROMELA interpreter provided as part of SPIN.

Communication in MABLE

In the version of MABLE described in [20], communication was restricted to `inform` and `request` performatives, the semantics of which were modelled on the corresponding FIPA performatives. However, this communication scheme rapidly proved to be too limiting, and has been significantly extended in the current version of MABLE. In particular, a user may use any kind of performative required: MABLE provides generic `send` and `receive` program instructions.

The abstract syntax of the message sending instruction is:

$$Ac \quad ::= \quad \texttt{send}(\texttt{CA}\ j\ \texttt{of}\ \varphi) \qquad /\text{* } j \in \mathit{AgId}, \varphi \in \mathit{wff}(\mathcal{MORA}) \text{ */}$$

(The sender of this message is not represented here, but is the agent executing the statement.) The basic meaning of the statement is that a message is sent to agent j using the communicative act CA: the content of the message is φ. (The keyword `of` is syntactic sugar only; it can be replaced by any identifier, and has no effect on the semantics of the program.)

Here is a concrete example of a MABLE `send` statement.

```
send(inform agent2 of (a == 10)
```

This means that the sender informs `agent2` that `a == 10`. For the moment, we will postpone the issue of the semantics of this statement; as we shall see below, it is possible for a programmer to define their own semantics, separately from the program itself.

The abstract syntax of the `receive` instruction is as follows.

$$Ac \quad ::= \quad \texttt{receive}(\texttt{CA}\ i\ \texttt{of}\ \varphi) \qquad /\text{* } i \in \mathit{AgId}, \varphi \in \mathit{VarId} \text{ */}$$

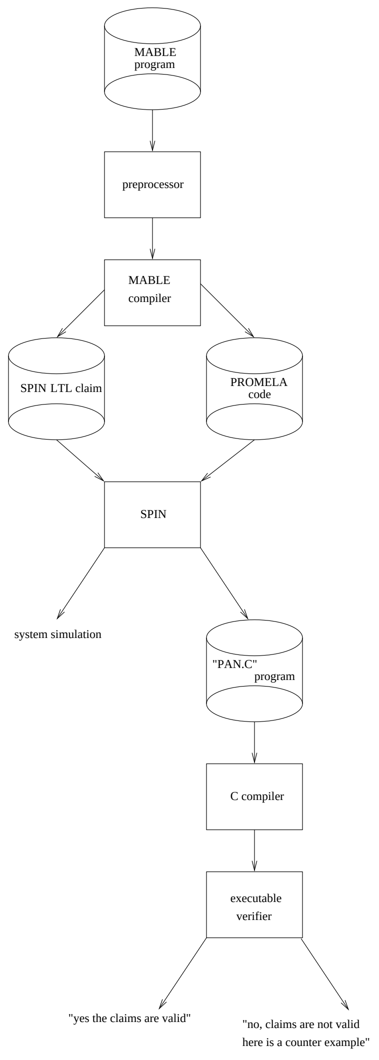

Figure 1: Operation of MABLE

As might be guessed, this means that the receiver receives a message from i for which the communicative act is CA and the message content is φ. Communication synchronous in the current version of MABLE, and so for this statement to succeed there must be a corresponding **send** by agent i.

A key component of the current instantiation of MABLE is that programmers can define their *own* semantics for communicative acts, separately from a program. Thus it is possible to explore the behaviour of the same program with a range of different semantics, and thus to investigate the implications of different semantics.

The basic model we use for defining semantics is a STRIPS-style pre-/post-condition formalism, in the way pioneered for the semantics of speech acts by Cohen and Perrault [2], and subsequently applied to the semantics of the KQML [3] and FIPA [5] languages. Thus, to give a semantics to performatives in MABLE, a user must define for every such communicative act a pre-condition and a post-condition. Formally, the semantics for a communicative act CA are defined as a pair $\langle CA_{pre}, CA_{post} \rangle$, where CA_{pre} is a condition (a MABLE predicate), and CA_{post} is an assertion. The basic idea is that, when an agent executes a **send** statement with performative CA, this message will not be sent until CA_{pre} is true. When an agent executes a **receive** statement with performative CA, then when the message is received, the assertion CA_{post} will be made true.

The MABLE compiler looks for performative semantic definitions in a file that is by convention named **mable.sem**. A **mable.sem** file contains a number of performative definitions, where each performative definition has the following structure:

```
i: CA(j, phi)
pre-condition
post-condition
```

where **i**, **j** and **phi** are the sender, recipient, and content of the message respectively, and **CA** is the name of the performative. The following lines define the pre-condition and post-condition associated with the communicative act CA. It is worth commenting on how these semantics are dealt with by the MABLE compiler when it generates PROMELA code.

With respect to the pre-condition, the above performative definition is translated into a PROMELA *guarded command* with the following structure.

```
    pre-condition -> send the message
```

The "**->**" is PROMELA's guarded command structure: to the left of **->** is a condition, and to the right is a program statement (an action). The semantics of the construct are that the process executing this statement will *suspend* (in effect, go to sleep) until the condition on the left hand side is true. When (more accurately, if) the condition becomes true, then the right hand side is "enabled": that is, it is ready to be executed, and assuming a fair process scheduler, will indeed be executed.

Notice that it is possible to define the pre-condition of a performative simply as "1", i.e., a logical constant for truth, which is always true; in this case, the send message part of the performative will always be enabled.

With respect to the post-condition, MABLE translates **receive** messages into PROMELA code with the following structure:

```
    receive message;
    make post-condition true
```

Thus once a message is received, the post-condition will be made true. Notice that post-conditions in a **mable.sem** file *do not* correspond to the "rational effect" parts of messages in FIPA semantics [4]; we elaborate on the distinction below.

Here is a concrete example of a **mable.sem** performative semantic definition:

```
i:inform(j,phi)
1
(believe j (intend i (believe j phi)))
```

This says that the sender of a message will always send an **inform** message directly; it will not wait to check whether any condition is true. It also says that when an agent receives an **inform** message, it will subsequently believe that the sender intends that the receiver believes the content.

Several examples of pre-conditions and post-conditions are given in section **??**. The use of semantics during the translation process is shown in Figure 2.

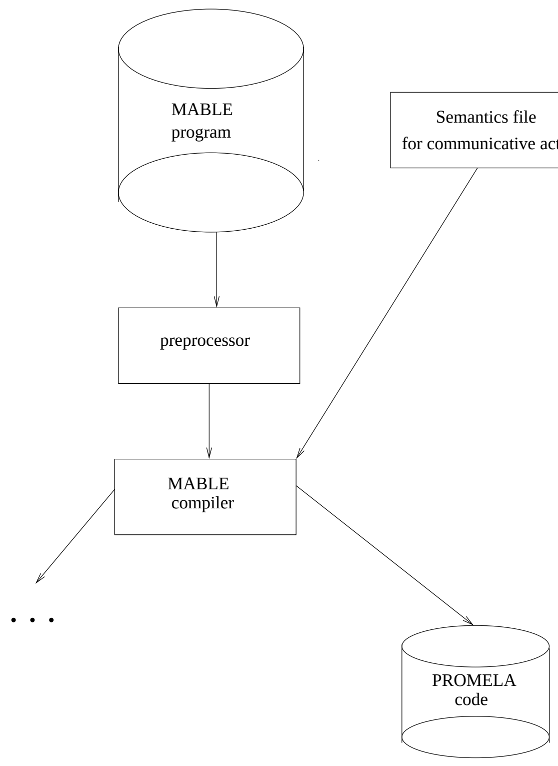

Figure 2: Operation of the MABLE system with the semantics file.

In summary, by disconnecting the semantics of a communicative act from a program that carries out such an act, we can experiment to see the effect that different kinds of semantics can have on the same agent. In the following section, we will see how this may be done in practice.

Claims

A key component of MABLE is that programs may be interspersed with *claims* about the behaviour of agents, expressed in $\mathcal{MORA}$, a subset of the $\mathcal{LORA}$ language introduced in [19]. These claims can then be *automatically* checked, by making use of the underlying SPIN model checker. If the claim is disproved, then a counter example is provided, illustrating why the claim is false.

A claim is introduced outside the scope of an agent, with the keyword `claim` followed by a $\mathcal{MORA}$ formula, and terminated by a semi-colon. The formal syntax of $\mathcal{MORA}$ claims is given in Figure 3. The language of claims is thus that of quantified linear temporal BDI logic, with the dynamic logic style `happens` operator, similar in intent and role to that in $\mathcal{LORA}$ [19]. Quantification is only allowed over finite domains, and in particular, over: agents (e.g., "every agent believes φ"); finite sets of objects (e.g., enumeration types); and integer number ranges. We will here give an overview of the main constructs of the claim language, focussing on those that are new since [20].

The goal of $\mathcal{MORA}$ (and also in fact of the whole MABLE framework) is that we should be able to verify whether programs satisfy properties of the kind expressed in BDI logics [14, 19]. To illustrate how $\mathcal{MORA}$ claims work, we here give some informal examples.

Consider the following $\mathcal{LORA}$ formula, which says that if agent a_1 believes the reactor failed, then a_1 intends that whenever a_2 believes the reactor failed (i.e., a_1 wants to communicate this to a_2).

$$(\mathsf{Bel}\ a_1\ \mathit{reactorFailed}) \Rightarrow (\mathsf{Int}\ a_1\ (\mathsf{Bel}\ a_2\ \mathit{reactorFailed}))$$

We can translate such a formula more or less directly into a $\mathcal{MORA}$ claim, suitable for use by MABLE. Consider the following:

```
claim []
  ((believe a1 reactorFailed) ->
  (intend a1 (believe a2 reactorFailed)));
```

The only noticeable difference is that, in the $\mathcal{LORA}$ formula, the intended interpretation is that we need to make the "whenever" explicit with the use of the temporal `[]` ("always") connective. The following $\mathcal{LORA}$ formula says that if some agent wants agent a_2 to believe that the reactor has failed, then eventually, a_2 will believe it has failed.

$$\forall i \cdot (\mathsf{Int}\ i\ (\mathsf{Bel}\ a_2\ \mathit{reactorFailed})) \Rightarrow \Diamond(\mathsf{Bel}\ a_2\ \mathit{reactorFailed})$$

This translates directly into the following $\mathcal{MORA}$ claim.

```
claim
 forall i : agent
 []((intend i (believe a2 reactorFailed))
   -> <>(believe a2 reactorFailed));
```

Thus far, the examples we have given illustrate features that were present in the version of MABLE documented in [20]; we now describe the main new feature of $\mathcal{MORA}$ claims, modelled on $\mathcal{LORA}$'s Happens construct [19, p.62]. In $\mathcal{LORA}$, there is a path expression of the form

$$(\mathsf{Happens}\ i\ \alpha)$$

which intuitively means "the next thing that happens is α". Thus, for example, the following $\mathcal{LORA}$ formula says that if agent a_1 performs the action of flicking the switch, then the reactor eventually hot.

$$(\mathsf{Happens}\ a_1\ \mathit{flick}) \Rightarrow \Diamond \mathit{reactorHot}$$

The current version of MABLE provides such a facility. We have a $\mathcal{MORA}$ construct

(happens *ag stmt***)**

where *ag* is the name of an agent and *stmt* is a MABLE program statement. This predicate will be true in a state whenever the next statement enabled for execution by agent *ag* is *stmt*. Consider the following concrete example.

```
claim
 []((happens a1 x = 10;)
   -> <>(believe a1 x==10));
```

This claim says that, whenever the next statement to be enabled for execution by agent `a1` is the assignment `x=10;` (notice that the semi-colon is part of the program statement, and must therefore be included in the `happens` construct), then eventually, `a1` believes that variable `x` has the value 10. (A single equals sign in MABLE is an assignment, while a double equals sign is the equality predicate.) As we will see below, the `happens` construct plays a key role in our approach to ACL compliance verification.

Before leaving this section, a note on how the `happens` construct is implemented by the MABLE compiler. The idea is to *annotate the model* that MABLE generates, with new propositions that will be set to be true in a given state whenever the corresponding agent is about to execute the corresponding action. To do this, the MABLE compiler passes over the parse tree of the MABLE program, looking for program statements matching those that occur in `happens` claims. Whenever it finds one, it inserts a program instruction setting the corresponding new proposition to true; when the program statement is executed, the proposition is set to false. The toggling of the proposition value is wrapped within PROMELA `atomic` constructs, to ensure that the toggling process itself does not alter the control flow of the generated system.

Although this process increases the size of the generated model, it does so only linearly with the number of `happens` constructs, and does not appear to affect performance significantly. Similarly, the pre-processing time required to insert new propositions into the model is polynomial in the size of the model and the number of `happens` claims.

4. VERIFYING ACL COMPLIANCE

We now demonstrate how MABLE can be used to verify compliance with ACL semantics. We begin with a running example that we will use in the following sections. The MABLE code is given in Figure 4. In this example, two agents have several beliefs and they simply send a message among themselves containing this belief. The selection of the message to be sent is done non deterministically through the `choose` statement. The insertion of these beliefs in agents' mental state is done through the `assert` statements. Beliefs correspond to conditions on values and differ from one agent to another one. After sending messages, agents wait for a message from the other agent. (Due to space restrictions, we do not give the PROMELA code that is generated by these examples.)

```
formula ::=
    forall IDEN ":" domain formula     /* universal quantification */
|   exists IDEN : domain formula       /* existential quantification */
|   any primitive MABLE condition      /* primitive conditions */
|   ( formula )                        /* parentheses */
|   (happens Ag stmt)                  /* statement is executed by agent */
|   (believe Ag formula)               /* agent believes formula */
|   (desire Ag formula)                /* agent desires formula */
|   (intend Ag formula)                /* agent intends formula */
|   [] formula                         /* always in the future */
|   <> formula                         /* sometime in the future */
|   formula U formula                  /* until */
|   ! formula                          /* negation */
|   formula && formula                 /* conjunction */
|   formula || formula                 /* disjunction */
|   formula -> formula                 /* implication */

domain ::=
    agent                              /* set of all agents */
|   NUMERIC .. NUMERIC                 /* number range */
|   { IDEN, ..., IDEN }                /* a set of names */
```

Figure 3: The syntax of $\mathcal{MORA}$ claims.

Verifying Pre-Conditions

Verifying pre-conditions consists means verifying that agents satisfy the pre-condition part of an ACL performative's semantics whenever they send the corresponding message. We will focus in this paper only on the `inform` performative; the cases for `request` and the like are similar.

Two approaches are possible for the pre-conditions: either agents are *sincere* (they only ever send an `inform` message if they believe its content), or else they are not (in which case they can send a message without checking to see whether they believe it). We can use MABLE's ACL semantics to define these two types of agents. Consider first the following `mable.sem` definition.

```
i:inform(j,phi)
(believe i phi)
(believe j (intend i (believe j phi)))
```

This says that the pre-condition for an `inform` performative is that the agent believes the content (`phi`) of the message. By defining the semantics in this way, an agent will only send the message if it believes it. (If the sender *never* believes the content, then its execution is indefinitely postponed.)

By way of contrast, consider the following `mable.sem` definition of the `inform` performative.

```
i:inform(j,phi)
1
(believe j (intend i (believe j phi)))
```

Here, the guard to the `send` statement is `1`, which, as in languages such as C, is interpreted as a logical constant for truth. Hence the guard will *always* succeed, and the message send statement will always be enabled, irrespective of whether or not the agent actually believes the message content. Notice that this second case is actually the more general one, which we would expect to find in most applications.

The next stage is to consider the process of actually checking whether or not agents respect the semantics of the language; of course, if we enforce compliance by way of the `mable.sem` file, then we would hope that our agents will always satisfy the semantics. But it is of course also possible that an agent will respect the semantics even though they are not enforced by the definition in `mable.sem`. (Again, this is in fact the most general case.)

For inform performatives, we can express the property to be checked in $\mathcal{LORA}$ [19] as follows:

$$\mathsf{A}\,\Box(\mathsf{Happens}\ i\ \mathit{inform}(j,\varphi)) \Rightarrow (\mathsf{Bel}\ i\varphi)$$

This formula simply says that, whenever agent i sends an "inform" message to agent j with content φ, then i believes φ. Now, given the enriched form of MABLE claims that we described above, we can directly encode this formula in $\mathcal{MORA}$, as follows:

```
claim
 []
  (
   (happens agent1
     send(inform agent2 of (a == 10));)
   ->
   (believe agent1 (a == 10))
  );
```

This claim will hold of a system if, whenever the program statement

```
send(inform agent2 of (a == 10));
```

is executed by `agent1`, then in the system state from which the `send` statement is executed, `agent1` believes that `a == 10`.

We can insert this claim into the system in Figure 4, and use MABLE to check whether it is valid. If we do this, then we find that the claim is indeed valid; inspection of the code suggests that this it what we expect.

```
int selection-agent1;
int selection-agent2;
agent agent1 {
  int inform-agent2;
  inform-agent2 = 0;

  selection-agent1 = 0;
  assert((believe agent1 (a == 10)));
  assert((believe agent1 (b == 2)));
  assert((believe agent1 (c == 5)));

  choose(selection-agent1, 1, 2, 3);
  if (selection-agent1 == 1) {
    print("agent1 -> a = 10\n ");
    send(inform agent2 of (a == 10));
  }
  if (selection-agent1 == 2) {
    print("agent1 -> b = 2\n ");
    send(inform agent2 of (b == 2));
  }
  if (selection-agent1 == 3) {
    print("agent1 -> c = 5\n ");
    send(inform agent2 of (c == 5));
  }

  receive(inform agent2 of inform-agent2);
  print("agent1 receives %d\n ", inform-agent2);

}

agent agent2 {

  int inform-agent1;
  inform-agent1 = 0;

  selection-agent2 = 0;
  assert((believe agent2 (d == 3)));
  assert((believe agent2 (e == 1)));
  assert((believe agent2 (f == 7)));

  choose(selection-agent2, 1, 2, 3);
  if (selection-agent2 == 1) {
    print("agent2 -> d = 3\n ");
    send(inform agent1 of (d == 3));
  }
  if (selection-agent2 == 2) {
    print("agent2 -> e = 1\n ");
    send(inform agent1 of (e == 1));
  }
  if (selection-agent2 == 3) {
    print("agent2 -> f = 7\n ");
    send(inform agent1 of (f == 7));
  }

  receive(inform agent1 of inform-agent1);
  print("agent2 receives %d\n", inform-agent1);

}
```

Figure 4: The base example.

Verifying pre-conditions implies as well that we check agents do not inform other agents about facts that they do not believe. Given the MABLE code presented in Figure 4, we have just to remove the line

```
assert((believe agent1 (a == 10)));
```

and then set the pre-condition of the `inform` to 1 (i.e., true) in the `mable.sem` file, and check the previous claim. Obviously, the claim is not valid since `agent1` informs `agent2` about something it does not believe.

Verifying Rational Effects

We consider an agent to be respecting the semantics of an ACL if it satisfies the specification defined by the pre-condition part of a message whenever it sends the message [18]. The rational effect part of a performative semantics define what the sender of the message wants to achieve by sending it; but of course, this does not imply that sending the message is sufficient to ensure that the rational effect is achieved. This is because the agents that receive messages are assumed to be autonomous, exhibiting control over their own mental state. Nevertheless, it is useful to be able to determine in principle whether an agent respects the rational effect part of an ACL semantics or not, and this is the issue we discuss in this section.

We will consider two cases in this section: *credulous* agents and *sceptical* agents. Credulous agents correspond to agents that always believe the information sent by other agents. We can directly define credulous agents in the following `mable.sem` file.

```
i:inform(j, phi)
(believe i  phi)
(believe j  phi)
```

This says that the recipient (j) of an `inform` message will always come to believe the contents of an `inform` message.

Sceptical agents are those that believe that the sender intends that they believe the information; they do not necessarily come to directly believe the contents of the message.

```
i:inform(j, phi)
(believe i  phi)
(believe j (intend i (believe j phi)))
```

We can directly define a $\mathcal{MORA}$ claim to determine whether or not an agent that is sent a message eventually comes to believe it.

```
claim []
 (
  (happens agent1
    send(inform agent2 of (a == 10));)
  ->
  <>(believe agent2 (a == 10))
 );
```

This claim is clearly valid for credulous agents, as defined in the `mable.sem` file given above; running MABLE with the example system immediately confirms this.

Of course, the claim may also be true for sceptical agents, depending on how their program is defined. We can directly check whether or not a particular sceptical agent comes to believe the message it has been sent, with the following claim:

```
claim
 []
 (
  (believe agent2
    (intend agent1
      (believe agent2 (a == 10))))
  ->
  <>(believe agent2 (a == 10))
 );
```

5. CONCLUSION

We have described extensions to the MABLE multiagent programming language and its associated logical claim language that make it possible to verify whether MABLE agents satisfy the semantics of ACLs. We illustrated the approach with a number of case studies. A key issue for future work is that of moving from the design level (which is what MABLE represents) to the implementation level, in the form of, for example, JAVA code. One possibility we are currently investigating is to enable MABLE to automatically generate JAVA code once a design has been satisfactorily debugged. Another interesting avenue for future work is investigating whether the MABLE framework might be used in the verification of other ACL semantics, such as Pitt's protocol-based semantics [12], or Singh's social semantics [17].

Acknowledgments This work was supported by the EC under project IST-1999-10948 (SLIE) and by the EPSRC under project GR/R27518.

6. REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.

[2] P. R. Cohen and C. R. Perrault. Elements of a plan based theory of speech acts. *Cognitive Science*, 3:177–212, 1979.

[3] T. Finin and R. Fritzson. KQML — a language and protocol for knowledge and information exchange. In *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence*, pages 126–136, Lake Quinalt, WA, July 1994.

[4] FIPA. Specification part 2 — Agent communication language, 1997. The text refers to the specification dated 23 October 1997.

[5] FIPA. Specification part 2 — Agent communication language, 1999. The text refers to the specification dated 16 April 1999.

[6] C. Guilfoyle, J. Jeffcoate, and H. Stark. *Agents on the Web: Catalyst for E-Commerce*. Ovum Ltd, London, April 1997.

[7] J. Y. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54:319–379, 1992.

[8] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall International: Hemel Hempstead, England, 1991.

[9] G. Holzmann. The Spin model checker. *IEEE Transaction on Software Engineering*, 23(5):279–295, May 1997.

[10] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1996.

[11] J. Mayfield, Y. Labrou, and T. Finin. Evaluating KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI Volume 1037)*, pages 347–360. Springer-Verlag: Berlin, Germany, 1996.

[12] J. Pitt and E. H. Mamdani. A protocol-based semantics for an agent communication language. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, Sweden, August 1999.

[13] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, pages 42–55. Springer-Verlag: Berlin, Germany, 1996.

[14] A. S. Rao and M. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–344, 1998.

[15] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[16] M. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, pages 40–49, December 1998.

[17] M. P. Singh. The intentions of teams: Team structure, endodeixis, and exodeixis. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI-98)*, pages 303–307, Brighton, United Kingdom, 1998.

[18] M. Wooldridge. Verifiable semantics for agent communication languages. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, pages 349–365, Paris, France, 1998.

[19] M. Wooldridge. *Reasoning about Rational Agents.* The MIT Press: Cambridge, MA, 2000.

[20] M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multiagent systems with MABLE. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002)*, pages 952–959, Bologna, Italy, 2002.