Model Checking for Multiagent Systems: The MABLE Language and its Applications

Michael Wooldridge* Marc-Philippe Huget+
Michael Fisher* Simon Parsons†

* Department of Computer Science, University of Liverpool
Liverpool L69 7ZF, United Kingdom
mjw | mdf@csc.liv.ac.uk

+ University of Savoie, ESIA-LISTIC
B.P. 806, 74016 Annecy CEDEX, France
Marc-Philippe.Huget@univ-savoie.fr

† Department of Computer & Information Science
Brooklyn College, City University of New York
2900 Bedford Avenue, Brooklyn NY 11210, USA
parsons@sci.brooklyn.cuny.edu

August 4, 2005

Abstract

We present MABLE, a fully implemented programming language for multiagent systems, which is intended to support the automatic verification of such systems via model checking. In addition to the conventional constructs of imperative programming languages, MABLE provides a number of agent-oriented development features. First, agents in MABLE are endowed with a BDI-like *mental state*: they have data structures corresponding to beliefs, desires, and intentions, and these mental states may be arbitrarily nested. Second, agents in MABLE communicate via ACL-like performatives: however, neither the performatives nor their semantics are hardwired into the language. It is possible to define the performatives and the semantics of these performatives independently of the system in which they are used. Using this feature, a developer can explore the design space of ACL performatives and semantics without changing the target system. Finally, MABLE supports automatic verification via model checking. Claims about the behaviour of a MABLE system can be expressed in a linear-time BDI-like logic, and the truth,

or otherwise, of these claims can be automatically determined. Following a description of the MABLE language and the language of MABLE claims, we present two case studies to illustrate the language and its use in the verification of multiagent systems. We then describe the key ideas underpinning the current implementation of MABLE. Finally, we survey related work, and discuss some avenues for future research.

1 Introduction

We present MABLE, a fully implemented programming language for multiagent systems [45], which is intended to support the automatic verification of such systems via model checking [4]. MABLE is novel in three key respects:

- Agents in MABLE have a *mental state* consisting of beliefs, desires and intentions; mental states may be nested, so that (for example), one agent is able to have beliefs about another agent's intentions.
- Agents in MABLE communicate using asynchronous message passing, in the style of the FIPA [11] and KQML [22] agent communication languages [7]. However, in MABLE, neither the agent communication language performatives themselves, nor their semantics, are hardwired into the language. Instead, it is possible for a developer to define both the performatives and the semantics of these performatives *independently* of the system in which they are used. In this way, a developer can explore the design space of ACL performatives and semantics without changing the target system itself.
- MABLE supports automatic verification via model checking [4]. Formal claims about the behaviour of a MABLE system can be expressed in a linear-time BDI-like logic, and the truth or otherwise of these claims can be automatically verified. Thus, in contrast to most logic-based agent programming languages, which perform reasoning at *run time*, reasoning about the correctness of a MABLE system is carried out at *design time* (we comment in more detail about the relationship of MABLE to other agent programming languages in section 5).

We emphasise that the MABLE language, as described in this paper, has been fully implemented. The implementation makes use of SPIN [16, 17], a freely available model checking tool for Linear Temporal Logic (LTL). The MABLE compiler takes, as input, a MABLE system together with associated claims about this system (expressed in a BDI-like logic), and generates, as output, both a representation of the MABLE system in PROMELA (the model specification language used by SPIN), and a translation of the BDI logic claims into the LTL logic used by SPIN. SPIN is then invoked, either to automatically verify the truth (or otherwise) of the claims, or else to simulate the execution of the MABLE system, using the PROMELA interpreter provided as part of SPIN.

The remainder of this paper is structured as follows. We begin by introducing the MABLE language, describe how claims can be made about MABLE programs using a BDI logic called $\mathcal{MORA}$, and show how these claims can be automatically verified using MABLE. We then present two detailed case studies, which illustrate the use of MABLE in the verification of multiagent systems. In the first case study, we demonstrate how MABLE can be applied to the problem of verifying that multiagent systems conform to the semantics of a particular agent communication language. While this is a well-known problem in the multiagent systems literature [41, 42, 26, 34], our work is, to the best of our knowledge, the first to apply model cheching techniques in this area. In the second case study, we present an implementation of the well-known Contract Net task allocation protocol [36, 35], and show how properties of this protocol can be verified using MABLE. We proceed to describe the operation of the MABLE compiler, and outline the key techniques used in its implementation. In section 5, we describe the relationship of MABLE to other research on agent programming languages and model checking for multiagent systems. Finally, we present some conclusions, and some pointers to future research.

Throughout the paper, we assume some familiarity with multiagent systems [45], model checking [4], and a basic understanding of conventional programming language design.

2 The MABLE Programming Language

MABLE is intended to be used as a language in which programmers can express and verify designs for multiagent systems. As such, one of the aims of MABLE is to provide a collection of constructs which closely resemble those used in conventional programming languages. However, a design requirement of MABLE was that it should be possible to automatically verify properties of systems using model checking: this requirement imposes some significant constraints on the facilities available to programmers in MABLE. For example, at an early stage of MABLE's development, the possibility of providing a JAVA-like object-oriented programming model was investigated. However, to provide such features would have necessitated the implementation of an object-oriented interpreter (similar to the JAVA virtual machine) in the modelling language of the target model checker, resulting in a dramatic blow-up in the size and complexity of models. The resulting state space explosion would almost certainly make the verification of systems impossible. For this reason, it was decided instead to provide a C-like imperative language, enriched by a number of key agent-oriented constructs. In particular, the key agent oriented features provided by MABLE over and above the basic system modelling facilities available in model checking systems such as SPIN [16, 17], SMV [24], and MOCHA [1] are as follows:

- Agents in MABLE have a *mental state* consisting of beliefs, desires and intentions.

- Agents in MABLE communicate using ACL performatives, and it is possible for a developer to define both the performatives and the semantics of these performatives independently of the system in which they are used.

In addition, MABLE provides "syntactic sugar" for many programming language features that are not provided as standard in most model checker system modelling languages (which tend to be rather low-level guarded command languages). In particular, MABLE provides the full range of iteration, sequence, and selection operations familiar from languages such as C and JAVA, C-like structure type declarations, and several high level synchronisation constructs. Note that we comment on the relationship of MABLE to other agent programming languages in section 5.

Over the past two decades, many logics and related formalisms have been proposed for representing and reasoning about multiagent systems, of which Rao and Georgeff's BDI logics are perhaps the best known [29, 44]. Ideally, then, we would like to take a BDI logic such as $\mathcal{LORA}$ (described in [44]) off the shelf, and develop verification tools that would allow us to determine whether or not systems implement specifications expressed in this logic. However, it is well-known that the link between such logics and implemented systems is informal at best. There is, in general, no systematic way of associating models for such logics with implemented systems: this is known as the problem of *computational grounding* [40, 43]. So, what we have done instead is to develop a slightly simplified and cut-down version of $\mathcal{LORA}$, known as $\mathcal{MORA}$, in which claims about systems can be expressed. We have then developed a mapping from this BDI-like logic to the Linear Temporal Logic used by the SPIN model checker; in this way, we can leverage existing model checking tools — and in particular, SPIN [16, 17] — to verify properties of MABLE systems.

In summary, then, a MABLE system consists of:

- a number of agent definitions and associated type and variable declarations, where each agent is programmed using the MABLE agent programming language;
- explicit semantics for the performatives used in the system;
- a number of formal claims about the system.

In the subsections that follow, we briefly describe these three elements. We begin with a survey of the agent programming language; we then describe the way in which the semantics of communication language performatives may be defined, and the use of claims in MABLE systems.

2.1 The Agent Programming Language

For a programmer, the core component of MABLE is of course the agent programming language. As noted above, this language is in essence a C-like imperative language, enriched by agent-oriented features. The concrete syntax of the conventional program

constructs in MABLE is based on that of C/JAVA, and so we will not give a detailed description here. Instead, we will give an overview of the main language features, focussing on those that are unique to MABLE.

Agent declarations and initialisation

Agents are declared via the `agent` keyword, followed by the agent's name (which must be unique), and the body of the agent. At startup, agents are invoked in the order in which they are declared; an agent terminates when it reaches the end of its code body. (Agents are not functions, and therefore do not `return` values; however, there is a `function` facility in MABLE, described below.) There is at present no way of invoking multiple copies of the same agent in MABLE, or of passing initialisation parameters to agents. However, it is possible for programmers to declare an explicit `init` section, which can be used for initialisation of system parameters. The `init` section is executed before any agent is invoked.

Beliefs, desires, and intentions

Perhaps the most obvious way in which MABLE differs from conventional programming languages is that the processes — agents — in a MABLE system have explicitly represented data structures corresponding to beliefs, desires, and intentions [29, 44]. These mental states can be nested, so that (for example) an agent can have beliefs about another agents intentions. Intuitively, an agent's beliefs are the information it has about the environment; these beliefs may be incorrect. An agent's desires and intentions come into play primarily when the agent is involved in communication.

In MABLE an agent's mental states are attitudes to the variables in the system. So, for example, agent `one` might have the belief that agent `two` intends that variable `x` has value greater than 10. Programmers can directly refer to an agent's mental state by means of *modal expressions*, or *modalities*. The intended meaning of the modality `(`m ag c`)` is that agent ag has attitude m (where m is `believe`, `desire`, or `intend`) towards the condition (predicate) c. The identifier ag must be the name of an agent in the system, and c must be a MABLE condition. The following is thus a legal modal expression in MABLE.

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

Suppose this expression is evaluated by `agent2`. Then it will "true" if `agent2` believes that `agent1` believes that `a == 10`. Mental states are implemented in MABLE as nested sets of facts (in the style of [18]): to evaluate this expression, `agent2` will check in its belief set, and inside this set will look for the set of facts representing `agent1`'s beliefs. If it finds that `a == 10` in this set (i.e., the set representing `agent2`'s beliefs about `agent1`'s beliefs), then the expression will evaluate to "true".

As modalities are themselves conditions, they may be arbitrarily nested. For example, the following is also a legal modal expression in MABLE:

```
(intend agent1 (believe agent2 (a == 10)))
```

Again, suppose this expression is evaluated by `agent1`: it will evaluate to "true" if `agent2` believes that `agent1` intends that `agent2` believes that `a == 10`.

In order to directly manipulate beliefs, desires, and intentions, MABLE provides `assert` and `retract` statements. These statements take a single argument — a condition — and behave rather like the PROLOG `assert` and `retract` predicates [5]. Thus, for example, consider the following MABLE statement.

```
assert((intend agent2 x == 10))
```

The effect of this statement is to make the agent executing it subsequently believes that `agent2` intends that variable `x` has the value 10.

An agent's mental state can also be modified in two other ways. First, an agent's beliefs can be changed by assignment and `observe` statements, as described in the following section. Second, communication actions may change an agent's mental state, as defined by in the performative semantics; we shall see how this works in section 2.2.

Types, variables, expressions, and assignments

MABLE supports C-style structure and array declarations, which may be composed in terms of integer and boolean data types. Variables in MABLE may be *local*, *shared*, or *global*. A local variable is private to an individual agent. A shared variable is declared outside an agent, and is visible to all agents in the system: all agents implicitly have access to shared variables, and moreover all agents can write to shared variables.

Like shared variables, global variables are also declared outside the scope of an agent. However, there is an important difference between global and shared variables. All agents implicitly know the value of shared variables; all agents have complete, correct, up-to-date *beliefs* about the value of shared variables. With global variables, however, the situation is slightly different. While all agents may still access global variables, *they must explicitly request access in order to discover their value*. They do this by executing a MABLE `observe` statement. The `observe` construct can thus be viewed as a *sensing action*. When an agent executes an `observe` instruction, its beliefs about the value of the variable it observes are synchronised with the true value of this variable. However, if the value of the variable is subsequently changed, then the agent will not necessarily be aware of this — its beliefs about the value of the variable may thus become "out of date". If an agent modifies the value of a global variable, then its beliefs about the value of this variable are similarly synchronised. Once again, however, its beliefs may become out of date if the value of this variable is changed by some other agent.

The syntax of variable declarations is broadly the same as C/JAVA. Expressions and assignment statements in MABLE also follow the conventions of C/JAVA; all the arithmetic operators that one would expect to find in an imperative language are present.

Conditional expressions

Conditional (boolean) expressions in MABLE may be constructed from expressions via the usual relational operators (`<`, `>`, `==`, ...). However, MABLE also permits conditions to contain modalities, as described above: in particular, belief, desire, and intention modalities.

Selection

MABLE contains the selection statements that one would expect from an imperative programming language — `if...else` and multi-way selection via `switch` statements. However, as noted earlier, the conditions in these constructs may contain belief, desire, and intention modalities. For example, suppose that agent `agent1` was executing the following statement.

```
if (intend agent2 (a == 10)) XYZ;
```

Then, in this case, `agent1` would execute statement `XYZ` if it believed that `agent2` intended that `a == 10`.

Loops

MABLE provides all the loop constructs found in C/JAVA (i.e., `for`, `while`, and `do`), and the syntax follows the conventions of these languages. There is an additional loop-like construct, which is not found in languages like C/JAVA: `await`. This construct implements an idle (non-busy) wait construct: it takes a single parameter, a condition, and the effect is that the agent executing the `await` is suspended until it believes the condition is satisfied.

Communication

MABLE provides two built-in communication primitives: `send` and `receive`. Their syntax is as follows:

`send(`p ag `of` c`);`
`receive(`p ag `of` c`);`

where p is the performative, ag is the name of an agent (the recipient of the message, in the case of `send`, the sender in the case of `receive`), and c is the message content, which must be a MABLE conditional expression. For example, the following is a syntactically acceptable `send` statement.

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

(Note that `of` is just syntactic sugar, which play no other role.) The following is also a legal `send` statement.

```
send(inform agent2 of (intend agent3 x == 10));
```

The effect of communication is to change the mental state of the recipient of the message. However, the actual effect that a message has is not defined within the program. It is defined externally, in the semantic definition file, as described below.

Note that message delivery is guaranteed, but is asynchronous: `receive` statements block until a message is available to be received, although `send` statements do not block. Broadcast message passing is not currently supported.

Synchronisation

In order to allow agents to synchronise their activities, MABLE provides facilities for enforcing mutual exclusion over critical sections of code. A MABLE system can contain an arbitrary number of *locks*, each of which is identified by a unique name. Sections of code can be wrapped in a `lock` statement, associated with a particular named lock. Only one agent can access a lock at any given time. When an agent comes across a locked section of code, it suspends until the associated lock is free, at which point it obtains the lock in an atomic operation, and enters the critical section; when it exits the code, the lock is released.

In addition, MABLE enables an agent to obtain exclusive access to a shared variable via the `read` construct. As long as an agent uses the `read` construct, it is impossible for other agents to access the variable locked with this construct. The lock is released when the agent exits the `read` block.

Functions

MABLE provides functions as a structuring mechanism for programs, and the syntax used for defining and invoking functions is again based on that of C. Functions may take arbitrary parameters, although at present may only `return` integer (`int`) values. All functions have global scope, and can be invoked either by agents or by other functions.

I/O

MABLE is intended primarily as a framework for model checking and, as such, there are critical limitations on the I/O facilities available in the language. Contemporary model checking techniques are focussed around *finite state* and hence *closed* systems. Thus it is not possible for a MABLE system to obtain input at run time from the outside environment. Where this is desired, a solution is to model the environment as an agent that provides appropriate input to other agents. However, a `print` statement is provided as a means to display output from MABLE.

Pre-processing

Before processing source code, the MABLE compiler runs the standard C pre-processor over files. This makes it possible to use all the pre-processor directives available in C:

- macro definitions, via the `#define` directive;
- textual file inclusion, via the `#include` directive;
- conditional compilation, via the `#if` ... `#endif` directive.

2.2 Communication in MABLE

A key component of the current version of MABLE is that programmers can define their *own* semantics for performatives, separately from a program in which these performatives are used. The formalism we use for defining semantics is a STRIPS-style pre-/post-condition model, in the way pioneered for the semantics of speech acts by Cohen and Perrault [6], and subsequently applied to the semantics of the KQML [19] and FIPA languages [11]. 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 is defined as a pair $\langle CA_{pre}, CA_{post} \rangle$, where CA_{pre} is a condition (a MABLE predicate), and CA_{post} is a condition to be asserted. 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.

By default, the MABLE compiler looks for performative semantics in a file that is 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 bound to the sender, recipient, and content of the message respectively, and `CA` is the name of the performative. The following two lines define the pre-condition and post-condition associated with the communicative act `CA`.

The way in which pre-conditions are used is as follows. Suppose an agent `agent1` executes the following statement

```
send(P agent2 of C)
```

where the semantics of the performative `P` are defined as follows.

```
i: P(j, phi)
pre
post
```

Then the agent `agent1` will *suspend* (i.e., enter a non-busy wait state) until the condition `pre` is believed to be true by `agent1`, at which point it will send the message. Notice that it is possible to define the pre-condition of a performative simply as "`1`", i.e., a logical constant for truth. In this case, the agent executing the `send` will never be suspended — the message will be sent immediately.

With respect to the post-condition, the idea is that once a message is received, the corresponding 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 [9]; 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.

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.

2.3 Claims

Another key component of MABLE is that agents may be augmented with formal *claims* about their behaviour. Claims are expressed in $\mathcal{MORA}$, a subset of the $\mathcal{LORA}$ BDI logic introduced in [44]. These claims can 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 1. 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 $\mathcal{LORA}$ [44]. The operators of $\mathcal{MORA}$ have the following intuitive meaning. First, any valid MABLE condition is an acceptable $\mathcal{MORA}$ formula, and thus it is possible to express conditions over all shared and global variables of a system. MABLE also supports the LTL operators of SPIN, as follows. First, `[]` ("always") is the *always in the future* operator: thus a formula `[]P` asserts that `P` will be true now

```
⟨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 1: The syntax of $\mathcal{MORA}$ claims.

(i.e., in the present state) and forever (i.e., in all future states). The <>P ("sometimes P") construct means "eventually, P will be true". In other words, P will either be true in the present state, or at some future state. (The <> construct does not assert the *unique existence* of such a state: it may be that P is several times in the future, or even that P is always true.) The U ("until") operator is a binary operator, and a formula P U Q asserts that P is true now, and will remain true until Q is true.

$\mathcal{MORA}$ supports quantification over finite domains, and in particular, over the following sets:

- agents (e.g., "every agent believes φ");
- finite sets of objects (e.g., enumeration types); and
- integer number ranges.

The `believe`, `desire`, and `intend` operators make it possible to make claims about agents' mental states. These constructs have the same interpretation in $\mathcal{MORA}$ claims as in conditionals, as described above.

To better understand how these constructs may be combined to make claims, consider the following informal examples.

First, suppose we want to express the fact that, whenever agent a_1 believes the reactor failed, then a_1 intends that a_2 believes the reactor failed (i.e., a_1 wants to communicate this to a_2).

We can express such a property directly as the following $\mathcal{MORA}$ claim.

```
claim
```

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

The outer `[]` is the temporal "always" operator, and ensures that this property is checked in every possible state that the system enters. Here, the variable `reactorFailed` is assumed to be boolean.

Next, suppose we want to say that if some agent wants agent a_2 to believe that the reactor has failed, then eventually, a_2 will believe it has failed.

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

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

Next, we describe the "`happens`" construct. Recall that the syntax of this construct is as follows:

`(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 that agent ag will perform 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 executed by agent `a1` is the assignment `x=10;`, then eventually, `a1` believes that variable `x` has the value 10. Notice that the semi-colon is part of the assignment program statement, and must therefore be included in the `happens` construct. Also recall that 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.

Finally, let us consider exactly how claims are checked by the MABLE compiler. Suppose that a system contains a single claim, φ, and that the programmer invokes the MABLE compiler signalling that this claim should be checked[1]. Then MABLE will systematically generate, (by means of the SPIN system), every possible computation $c = s_0, s_1, s_2, \ldots$ of the system. Each computation c corresponds to a *model* for $\mathcal{MORA}$, and the claim φ will either be true or false when interpreted in this model. So, for every computation c, MABLE will check whether this computation satisfies φ; if

[1]The default behaviour of the MABLE compiler is to ignore claims; a user indicates to the compiler that claims should be checked by means of a command line argument.

MABLE ever encounters a computation c such that $c \not\models \varphi$, then MABLE halts, and reports c as a counterexample to the claim φ. If no such computation is found, then MABLE (or more accurately, SPIN!) will continue until it has exhaustively examined the entire space of possible computations.

3 Two Case Studies

This section presents two detailed case studies. The first case study demonstrates how MABLE can be used to verify that agents correctly implement the semantics of an agent communication language [41]. In the second case study, we show how MABLE can be used to implement the Contract Net protocol [36, 36], and we show how properties of this protocol can be established via model checking.

3.1 Verifying Compliance with respect to ACL Semantics

In this section, we will show how conformance to the pre-condition and rational effect parts of ACL semantics can be verified with MABLE. We also show how, by varying the semantics of performatives, we achieve different results for the same agent programs. We begin with a brief introduction to the ACL verification problem.

The need for agents to be able to inter-operate has led to the development of several standardised *agent communication languages* (ACLs) [22, 10]. 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 — recognises 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" [10], 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 (*verification*) is important. However, the problem of compliance testing is not actually given a concrete definition by FIPA, and no indication is given of how it might be done.

In [41], 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 [10, p25]:

$$\begin{array}{l}\langle i, inform(j,\varphi)\rangle \\ \quad \text{FP:} \quad B_i\varphi \wedge \neg B_i(Bif_j\varphi \vee U_j\varphi) \\ \quad \text{RE:} \quad B_j\varphi\end{array} \tag{1}$$

Here $\langle i, inform(j,\varphi)\rangle$ 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 components 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., [14]); 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 [41] 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.

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 [26]: the idea here 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 here being that an ACL semantics should be understood in terms of the observable, verifiable changes in social state (the relationships between agents) that use of a performative causes [34].

We begin with a running example that we will use in the following sections to illustrate the approach. The example employs the "`inform`" performative, which is one of the two key performatives in the FIPA framework [11]. (The other is "`request`", which can be dealt with using the same techniques.) The MABLE code for this example is given in Figures 2 and 3. Two agents have several beliefs and they simply send messages among themselves communicating these beliefs. The selection of the message to be sent is carried out non-deterministically, via the `choose` construct. The insertion of these beliefs in agents' mental state is done through the `assert` statements.

```
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);
}
```

Figure 2: The base example (agent 1).

```
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 3: The base example (agent 2).

Verifying Performative Pre-Conditions

Verifying pre-conditions means verifying that agents satisfy the pre-condition part of an ACL performative's semantics whenever they send the corresponding message.

There are essentially two possibilities with respect to 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 pre-condition test 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 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 FIPA-style `inform` performatives, the property we want is 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 given in Figures 2 and 3, 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 is what we expect.

Verifying pre-conditions also implies that we ensure agents do not inform other agents about facts that they do not believe. In our running example, we simply have 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. The claim is now not valid, as `agent1` informs `agent2` about something it does not believe.

Verifying Performative 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 [41]. The rational effect part of a performative's semantics defines what the sender of the message wants to achieve by sending it; but 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: *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 via 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, but 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))
 );
```

3.2 The Contract Net Protocol

In the section, we will show how the well-known Contract Net protocol can be implemented using MABLE [36, 35], and then demonstrate how properties of this implementation may be verified using MABLE claims.

The Contract Net Protocol was proposed by Smith [36, 35] as a mechanism for task allocation in distributed problem solving systems. The idea of this protocol is that one agent (the initiator of the interaction) has a task to carry out, but requires cooperation for this task — either because the task requires resources that are unavailable to the initiator, or else because a cooperative solution will be preferred to a non-cooperative one. The initiator takes the role of *task manager* and broadcasts an announcement of the task to other agents. In general, the task announcement specifies the properties of the task — quality of service parameters, and any other information that a potential

bidder may require to determine whether or not to submit a bid to carry out the task. In our implementation, the task announcement defines the skills required to solve the task (its "weight").

Agents receiving a task announcement have several choices. They can either submit a bid for the task (e.g., specifying a price for carrying the task out), or else they can choose not to bid. When the task manager has the answers of the bidders, it can choose a bidder, to whom it awards the task.

The implementation of the Contract Net protocol in MABLE represents about 250 lines of code. It contains three agents: the task manager and two bidding agents. Additionally two functions are declared.

```
/* use for the loop to set type skills and reward for each bidder */
int i, value;
// used in the "decision" function
int a;
/* used in function select */
int max-value;
int accepted;
int index;
int index-reward;
int ca-bidder[2]; int value-bidder[2];
int number-accepted;
int accept-bidder[2];
int rank;

/* structures containing the maximum type possible
and the minimum expected reward for a task for each bidder */
struct capability {
    int max-type;
    int min-reward;
};
struct capability capabilities[2];
/* structure of a task */
struct task {
    int id;
    int type;
    int reward;
};
/* since it is not possible to send structures in messages,
the task is declared globally*/
struct task one-task;
```

Figure 4: Contract Net Protocol Declarations.

The variable declarations for the system are shown in Figure 4. The `init` section for the system is shown in Figure 5. In this section, we first define the task, then we define bidders' parameters: the maximum size of task they can perform, and their expected reward for a task. These values are set non-deterministically, through the `choose` construct.

The implementation of the task manager is given in Figure 6. The task manager first informs the two bidders that a task has to be performed, and then waits for an answer. In our model, bidders are obliged to answer either with an acceptance or a

```
init {
     /*set the task*/
     one-task.id = 1;
     one-task.type = 10;
     one-task.reward = 5;
     /*set bidders' competences, type and price are set at random*/
     i = 0;
     while (i < 2) {
           choose(value, 8, 10, 15);
           capabilities[i].max-type = value;
           choose(value, 2, 5, 7, 8, 10);
           capabilities[i].min-reward = value;
           i = i + 1;
     }
}
```

Figure 5: Initialisation for the Contract Net.

rejection. As soon as the task manager has received all answers, it selects at most one bidder to process the task. The `TaskManager` uses the `select` function to choose which agent to award the task to. Finally, the task manager sends a message to the successful bidder.

The implementation of bidding agents is shown in Figure 7. The first action of the bidders is to wait for the task announcement. Then, they call the `decision` function to determine if they are able to do the task. The decision is made on the basis of the task type and the reward.

We have two functions in the Contract Net implementation: the `select` function, used by the task manager to select a bidder to perform the task; and the `decision` function, used by bidders to ascertain if they are able to perform the task. The implementation of the `select` function is shown in Figure 8, while the `decision` function is shown in Figure 9.

The `decision` function is used by the bidders to know if they are able to perform the task. This decision is determined by the task type and the reward. If the task type is within their capabilities, and if the reward is greater or equal to their request, then they accept the task. If the task type is beyond their capabilities, they refuse the task. If the reward is less than the one expected, they accept the task but only for their requested reward.

Running the example

After writing the MABLE code, designers can execute the system: the following output was generated by MABLE when it was invoked with the Contract Net example in simulation mode[2].

```
Bidder 0 launched!
TaskManager launched!
```

[2]Note that this is just one possible run among several.

```
/*definition of the task manager agent, it is responsible to advertise
the task, to select a bid and to request the task to be performed*/
agent TaskManager {
      /*return of the function accept, if the value is 0, there is no
      clear accept, else the value corresponds to the id of the
      bidder, 1 for Bidder 1, etc.*/
      int result;
      /*these values store the message content of bidders' answer*/
      int value1, value2;
      /*TaskManager needs to use these variables since it does not
      know in advance what messages will be sent:  accept or refuse */
      int ca-Bidder1, ca-Bidder2;
      print("TaskManager launched!");
      /*task advertisement*/
      send(inform Bidder1 of one-task.id);
      send(inform Bidder2 of one-task.id);
      /*collecting the answers from bidders; ca-BidderX contains
      either accept or refuse, that is to say 1 or 2*/
      value1 = 0; value2 = 0;
      receive(ca-Bidder1 Bidder1 of value1);
      receive(ca-Bidder2 Bidder2 of value2);
      /*TaskManager has to select a bidder*/
      ca-bidder[0] = ca-Bidder1; value-bidder[0] = value1;
      ca-bidder[1] = ca-Bidder2; value-bidder[1] = value2;
      result = select();
      switch(result) {
            case 0:  send(request Bidder1 of one-task.id);
            case 1:  send(request Bidder2 of one-task.id);
      }
}
```

Figure 6: The Task Manager Definition.

```
Bidder 1 launched!
Bidder 1 refuses the task, too heavy!
Bidder 0 accepts the task but with a different reward
quitting...
quitting...
Bidder 0: I do the task
quitting..
7 processes created
```

Model Checking the Contract Net Protocol

Having implemented the Contract Net in MABLE, it is natural to then use MABLE's verification capabilities to check the implementation. We will just give two properties that may be checked:

1. when the task is advertised, eventually it will be awarded to some agent;
2. when the task is advertised, eventually it will be performed at a different reward.

As stated in section 2.3, properties have to be expressed as claims to be checked. The first property gives the following claim:

```
agent Bidder1 {
      /*this variable contains the id of the task*/
      int task-advertised;
      /* the result whether the bidder accepts the bid or
          not given the constraints
      1 corresponds to a clear accept, 2 to a rejection, > 2 the
      new proposed reward */
      int result;
      print("Bidder 0 launched!");
      /*waiting for the task advertisement*/
      receive(inform TaskManager of task-advertised);
      /*the bidder decides if it is able to do the task*/
      result = decision(0);
      if (result == 1) {
         send(accept TaskManager of task-advertised);
      }
      if (result == 2) {
         send(refuse TaskManager of task-advertised);
      }
      if (result > 2) {
         send(accept TaskManager of result);
      }
      /*waiting for a possible answer from TaskManager*/
      if (result == 1 || result > 2) {
         receive(request TaskManager of task-advertised);
         /*if the bidder receives a message, it means it has
      to perform the task */
         print("Bidder 0:  I do the task");
      }
}
```

Figure 7: The Bidder Definition.

```
claim
  [](((happens TaskManager
              send(inform Bidder1 of one_task.id);)
  ->
  <> exists ag : agent
  (happens ag
           receive(request TaskManager
                           of task_advertised);));
```

The second property corresponds to the following claim.

```
claim
  [](((happens TaskManager
              send(inform Bidder1 of one_task.id);)
  -> <>(one_task.reward < max_value));
```

It took about six minutes on a PC with an Intel Pentium III 500MHz processor and 256Mb RAM to verify each of these results.

```
function int select() {
         accepted = 0;
         max-value = 65535;
         index = 0;
         index-reward = 0;
         number-accepted = 0;
         rank = 0;
         while (index < 2) {
            if (a-bidder[index]==1 && value-bidder[index]==one-task.id) {
               /*this is a clear accept*/
               accept-bidder[number-accepted] = index;
               number-accepted = number-accepted + 1;
               accepted = 1;
               index = index + 1;
            }
            else {
               if (!accepted && ca-bidder[index] == 1 &&
            value-bidder[index] != one-task.id) {
                  if (max-value > value-bidder[index]) {
                  max-value = value-bidder[index];
                  index-reward = index;
                  }
                  index = index + 1;
               }
               else {
                  if (ca-bidder[index] == 2) {
                     index = index + 1;
                  }
               }
            }
         }
         /*clear accept */
         if (accepted) {
            if (number-accepted == 1) {
               rank = accept-bidder[0];
               return rank;
            }
            if (number-accepted == 2) {
               choose(rank, 0, 1);
               rank = accept-bidder[rank];
               return rank;
            }
         }
         else {
            if (max-value != 65535) {
               return index-reward;
            }
            else {
                 return -1;
            }
         }
}
```

Figure 8: The Select Function.

```
/*function that decides if bidders are able to do the task*/
function int decision(bidder) {
   /*we need to store the field of the structure since
   they are not available in conditions*/
   a = capabilities[bidder].max-type;
   /*first, we test if the type is much more than accepted for this
   bidder*/
   /*in this 'if', bidders accept or accept with a greater reward*/
   if (a >= one-task.type) {
      /*then, we test if the reward is greater than the one expected*/
      a = capabilities[bidder].min-reward;
      if (a <= one-task.reward) {
         print("Bidder %d accepts the task", bidder);
         return 1;
      }
      else {
      /*the reward is less than the one expected, the bidder proposes
   a new reward to the TaskManager*/
         print("Bidder %d accepts the task with different reward", bidder);
         return a;
      }
   }
   /*in this case, bidders refuse the task, too heavy*/
   else {
      print("Bidder %d refuses the task, too heavy!", bidder);
      return 2;
   }
}
```

Figure 9: The Decision Function.

4 The MABLE Compiler

In this section, we give a brief overview of the way in which the MABLE compiler works. The compiler translates MABLE systems into a form that can be processed by the SPIN model checker [16, 17]. The way in which the MABLE compiler interacts with SPIN is illustrated in Figure 10.

There are four key components to the MABLE compiler: the way in which individual agents and their control constructs (e.g., loops) are translated to PROMELA; the way in which belief-desire-intention states are implemented; the way in which $\mathcal{MORA}$ claims are dealt with; and the way in which performative semantics are dealt with.

Agents and Basic Control Structures

Dealing with the basic MABLE control constructs is straightforward. Although PROMELA is a relatively low-level language, it is straightforward to map MABLE's control constructs into those provided by PROMELA. Agents in MABLE are implemented as processes (`proctypes`) in PROMELA; additional PROMELA initialisation code is generated to (automatically) start agents simultaneously.

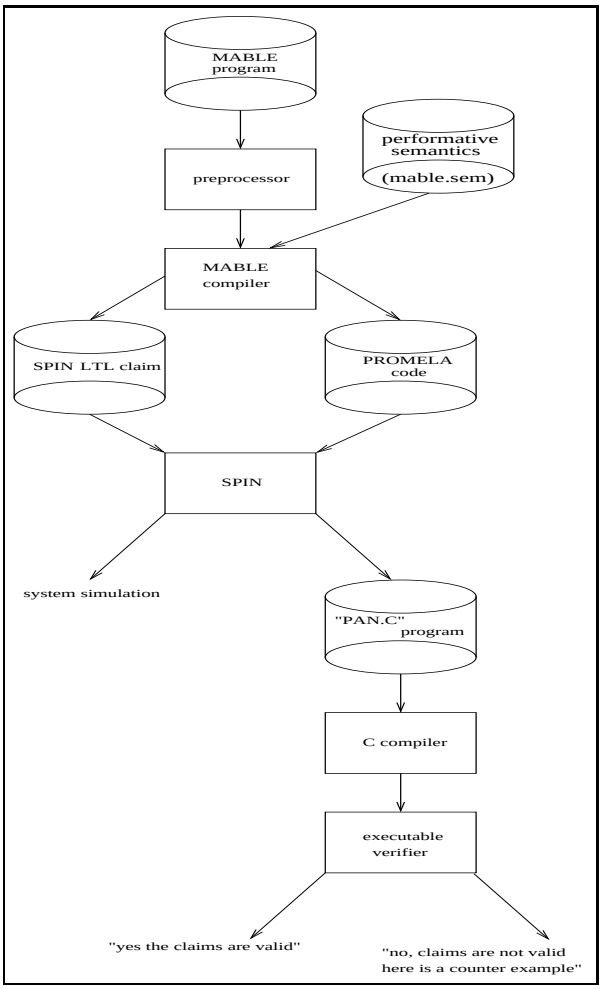

Figure 10: Operation of the MABLE system.

Mental States

More interesting is the way that mental states are dealt with. The idea is to model these as finitely nested data structures (in the style of [18, 2]). Predicates are represented by *propositional abstraction*: where a predicate appears in the context of a modality, a new proposition symbol is introduced to represent this predicate. This new proposition is then used in the BDI data structures. A key disadvantage of this approach is that it causes a blow-up in the size of the state space. For this reason, nested mental states are only generated on an "as needed" basis.

Claims

To implement claims, we need to map $\mathcal{MORA}$ formulae into the LTL form accepted by SPIN. In this mapping, we need to deal with a number of features that are not supported directly by LTL:

- Quantifiers are removed by *expansion*. Quantification is over finite domains, and so any quantified formula can be rewritten into a quantifier-free formula by expanding universal quantification into a conjunction, and existential quantification into a disjunction.
- BDI modalities are removed by replacing them with predicates about the corresponding data structures in the implemented system.
- Predicates are removed by propositional abstraction: each predicate is replaced by proposition, the truth of which is bound to the predicate it replaces.
- To deal with `happens` operators, we insert new code into the program itself, flagging the occurrence of statements that occur in `happens` operators. These flags can then be referred to in the LTL formulae generated by MABLE. Suppose we have an operator `(happens ag s)` occurring in a claim. First, MABLE replaces this operator in the claim with a new proposition, say `p`. The MABLE compiler then passes over the parse tree of the MABLE program for agent `ag`, looking for the statement `s` in the program code. Whenever it finds `s` in the parse tree, it inserts a new statement into the program immediately before `s`, setting the corresponding new proposition `p` to true; and following the statement `s`, another new program statement is inserted, setting the proposition `p` to false. The toggling of the proposition `p` is wrapped within PROMELA `atomic` constructs, to ensure that the toggling process itself does not alter the control flow of the generated system. In this way, the truth of the proposition `p` indicates that the next statement to be executed by `ag` is `s`.

The end result is a propositional LTL formula, suitable for input to the SPIN model checker, together with a list of predicates and the names of the propositions with which they were replaced. Together with the generated PROMELA code, these can be fed directly into SPIN for checking.

Performative Semantics

Recall that it is possible to define the semantics of performatives separately from the MABLE system itself, using the `mable.sem` file. The MABLE compiler looks for such a file containing 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. These semantic definitions are dealt with as follows.

With respect to the pre-condition, suppose that a particular agent contained a `send` statement with the performative `CA`. Then this `send` would be 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 this 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.

With respect to the post-condition, suppose an agent contained a `receive` statement with the performative `CA`. Then this `receive` statement would be translated into PROMELA code with the following structure.

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

Thus once a message is received, the post-condition will be asserted.

5 Related Work

In recent years, a number of logic-oriented multi-agent programming languages have been developed, which attempt to bring logics of rational agency somewhat closer to programming languages. In this section, we will briefly consider the relationship of MABLE to this work.

Perhaps the best-known multiagent language is AGENT0 [33]. Developed by Shoham in the late 1980s, this was the first language to be explicitly referred to as

an "agent-oriented programming" language; additionally, Shoham was keen to link AGENT0 to a multimodal logic of rational agency [39]. The logic was clearly intended to provide a logical semantics for the language, although the precise details were never made formal. The programming model for AGENT0 was that of rule-based systems: one programmed an agent in terms of a set of commitment rules, which defined how an agent formed and discharged commitments to action. Agents communicated with one-another using three performatives: `request`, `unrequest`, and `inform`. The semantics of these performatives was not really hardwired into the language: a programmer defined their effects by writing rules to handle them.

A number of later languages were developed in the AGENT0 mould. The PLACA language, adopting a rule-based programming model very similar to that of AGENT0, was intended to overcome a number of deficiencies of AGENT0, such as the ability of agents to explicitly plan how to meet their commitments [38].

Rao's AGENTSPEAK was another influential agent-oriented programming language [28]. Agents in AGENTSPEAK are programmed by defining a set of plans (somewhat like Shoham's commitment rules), which are executed both reactively (plans are invoked in response to events in the environment), and pro-actively (plans can be explicitly invoked by other plans). This programming model is essentially a distillation of Georgeff and Lansky's Procedural Reasoning System (PRS) [13], a reactive planning framework developed in the mid-1980s that subsequently formed the basis of several other agent-oriented programming systems, notably DMARS [8]. Rao and Georgeff developed a number of BDI logics, which, like Shoham's logics, were ultimately intended to provide a semantics to AGENTSPEAK and the PRS [29]. However, as with Shoham's language, while there was plenty of intuition about how the logics related to the programming language [30], the precise relationship between language and logic was never made formal. As a consequence, one could never really claim that a formula of the logic expressed a property of a system; and as a consequence, one could never verify whether the formula represented a property that was true or false of a given system. It is worth noting that Rao and Georgeff developed preliminary model checking techniques for BDI logics [31], although because the relationship between the logic and the PRS/AGENTSPEAK was never made precise, these model checking techniques could not be deployed to verify PRS/AGENTSPEAK systems [40, 43]. (It is worth noting that at the time of writing, work is underway to develop techniques for model checking AGENTSPEAK systems [3].) Note that Hindriks' 3APL language was a direct descendant of AGENSPEAK [15]. In 3APL, agents are also programmed using a rule-like model; the semantics and proof theory of 3APL were developed in some detail.

GOLOG [21, 32] and its multiagent sibling CONGOLOG [20] represent another rich seam of work on logic-oriented approaches to programming rational agents. Essentially, GOLOG is a framework for executing a fragment of the situation calculus; the situation calculus is a well known logical framework for reasoning about action [23]. Put crudely, writing a GOLOG program involves expressing a logical theory of what action an agent should perform, using the situation calculus; this theory, together with

some background axioms, represents a logical expression of what it means for the agent to do the right action. Executing such a program reduces to constructively solving a deductive proof problem, broadly along the lines of showing that there is a sequence of actions representing an acceptable computation according to the theory [32, p.121]; the witness to this proof will be a sequence of actions, which can then be executed.

A closely related approach is that of the METATEM paradigm [12]. In METATEM, an agent is programmed by giving it a temporal logic specification of the behaviour it should exhibit, where this specification is a conjunction of *past* $\Rightarrow$ *future* rules. The process of executing the specification corresponds to doing a constructive proof of the satisfiability of the program formula, where the model being constructed is built in part by the agent, and part by the environment. Another somewhat related language is the IMPACT framework of Subrahmanian et al. [37]. IMPACT is a rich framework for programming agents, which draws upon and considerably extends some ideas from logic programming. Agents in IMPACT are programmed by using rules that incorporate deontic modalities (permitted, forbidden, obliged [25]). These rules can be interpreted to determine the actions that an agent should perform at any given moment [37, p.171].

A common feature of all the languages mentioned in this section is that the reasoning used to determine the action to perform takes place at *run time*. Moreover, as we are in all cases essentially executing a logical formula, the issue of verification does not really arise — for example, we can be assured that any execution trace of a METATEM program is a model of the program formula. Indeed, this is one of the key arguments in favour of using an executable logic framework: we can be assured that a logical program will execute according to its semantics.

There are, however, several disadvantages to deciding what action to perform by reasoning at run time. The most obvious of these is that reasoning is generally computationally costly: see, for example, the complexity results associated with algorithms for the IMPACT framework [37, pp.399–460]. A more subtle problem is that requiring a programmer to express a program in the language of logic (be it situation calculus, temporal logic, or deontic logic) is often not desirable. Ideally, we want to be able to let the programmer use their most preferred programming tools, and then verify their work, rather than imposing a programming regime on them. In this sense, we believe, MABLE is closer to the reality of everyday programming. It provides constructs corresponding to those that programmers everywhere are familiar with, enriched with agent-oriented constructs. As verification is done at design time, the issue of run-time reasoning (and all the potential difficulties it entails) does not arise. The disadvantage of the MABLE approach is of course that we lose the elegant logical semantics associated with directly executing logical formulae; but we argue that as we can directly verify MABLE systems, this is not a major issue.

6 Conclusions

In this paper, we have described the MABLE language for multiagent systems. This fully-implemented language supports the development of agents in an imperative programming language, enriched by some features from the agent-oriented programming paradigm. However, the most important (and novel) feature of MABLE is that it supports the automatic verification of MABLE systems via model checking. A designer can formally express the requirements of MABLE systems as formulae of linear-time BDI logic, and MABLE is capable of automatically verifying whether or not the system does or does not satisfy these requirements. Another novel feature is that although the key communication mechanism in MABLE is asynchronous message passing in the style of FIPA and KQML, MABLE does not dictate a semantics for the performatives used in communication. Instead, a designer can explicitly define the semantics of performatives separately from a system and, in this way, can explore the behaviour of the same system for a range of performative semantics. Combining this feature with the model checking capabilities of MABLE, it becomes possible to automatically verify compliance to agent communication language performatives — a problem of some interest to the agent communication language community.

There are a number of obvious avenues for future research. First, we hope to further extend the language and its model checking facilities. For example, it would be useful to add features such as true unification to the language, and other similar reasoning features. However, the addition of such features will inevitably lead to a (further) blow-up in the state space of the generated system. For this reason, we intend to study the possibility of *automatic abstraction* of MABLE systems: essentially, stripping MABLE systems down to their leanest possible representation. Another issue we are pursuing is that of automatically generating JAVA code from MABLE systems. For example, suppose we have a MABLE system that we have verified complies with the semantics of the FIPA language. Then automatically generating JAVA code that implements the FIPA performatives via the JADE implementation of the FIPA language [27], we can plausibly (if not entirely accurately) claim that the resultant JAVA system respects the FIPA semantics.

References

[1] R. Alur, L. de Alfaro, T. A. Henzinger, S. C. Krishnan, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Taşiran. MOCHA user manual. University of Berkeley Report, 2000.

[2] M. Benerecetti, F. Giunchiglia, and L. Serafini. A model checking algorithm for multiagent systems. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V (LNAI Volume 1555)*. Springer-Verlag: Berlin, Germany, 1999.

[3] R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking agentspeak. In *Proceedings of the Second International Conference on Autonomous Agents and Multiagent Systems (AAMAS-03)*, Columbia University, NY, USA, July 2003.

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

[5] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag: Berlin, Germany, 1981.

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

[7] F. Dignum and M. Greaves, editors. *Issues in Agent Communication (LNAI Volume 1916)*. Springer-Verlag: Berlin, Germany, 2000.

[8] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV (LNAI Volume 1365)*, pages 155–176. Springer-Verlag: Berlin, Germany, 1997.

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

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

[11] FIPA. The foundation for intelligent physical agents, 2001. See `http://www.fipa.org/`.

[12] M. Fisher. A survey of Concurrent METATEM — the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic — Proceedings of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Berlin, Germany, July 1994.

[13] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.

[14] 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.

[15] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–402, 1999.

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

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

[18] K. Konolige. *A Deduction Model of Belief*. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1986.

[19] Y. Labrou and T. Finin. Semantics and conversations for an agent communication language. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 584–591, Nagoya, Japan, 1997.

[20] Y. Lésperance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI Volume 1037)*, pages 331–346. Springer-Verlag: Berlin, Germany, 1996.

[21] 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.

[22] 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.

[23] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[24] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers: Dordrecht, The Netherlands, 1993.

[25] J.-J. Ch. Meyer and R. J. Wieringa, editors. *Deontic Logic in Computer Science — Normative System Specification*. John Wiley & Sons, 1993.

[26] 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.

[27] Agostino Poggi and Giovanni Rimassa. Adding extensible synchronization capabilities to the agent model of a FIPA-compliant agent platform. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop AOSE-2000 (LNCS Volume 1957)*, pages 307–322. Springer-Verlag: Berlin, Germany, 2001.

[28] 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.

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

[30] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-92)*, pages 439–449, 1992.

[31] A. S. Rao and M. P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 318–324, Chambéry, France, 1993.

[32] R. Reiter. *Knowledge in Action*. The MIT Press: Cambridge, MA, 2001.

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

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

[35] R. G. Smith. The contract net protocol. *IEEE Transactions on Computers*, C-29(12), 1980.

[36] R. G. Smith. *A Framework for Distributed Problem Solving*. UMI Research Press, 1980.

[37] V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. The MIT Press: Cambridge, MA, 2000.

[38] S. R. Thomas. The PLACA agent programming language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 355–369. Springer-Verlag: Berlin, Germany, January 1995.

[39] S. R. Thomas, Y. Shoham, A. Schwartz, and S. Kraus. Preliminary thoughts on an agent description language. *International Journal of Intelligent Systems*, 6:497–508, 1991.

[40] M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144(1):26–37, February 1997.

[41] 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.

[42] M. Wooldridge. Verifying that agents implement a communication language. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 52–57, Orlando, FL, July 1999.

[43] M. Wooldridge. Computationally grounded theories of agency. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-2000)*, pages 13–20, Boston, MA, 2000.

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

[45] M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002.