Model Checking Multi-Agent Systems with MABLE[*]

Michael Wooldridge Michael Fisher Marc-Philippe Huget Simon Parsons
Department of Computer Science, University of Liverpool
Liverpool L69 7ZF, United Kingdom
{mjw,mdf,mph,sp}@csc.liv.ac.uk

ABSTRACT

MABLE is a language for the design and automatic verification of multi-agent systems. MABLE is essentially a conventional imperative programming language, enriched by constructs from the agent-oriented programming paradigm. A MABLE system contains a number of agents, programmed using the MABLE imperative programming language. Agents in MABLE have a mental state consisting of beliefs, desires and intentions. Agents communicate using `request` and `inform` performatives, in the style of the FIPA agent communication language. MABLE systems may be augmented by the addition of formal claims about the system, expressed using a quantified, linear temporal belief-desire-intention logic. MABLE has been fully implemented, and makes use of the SPIN model checker to automatically verify the truth or falsity of claims.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Software Engineering; I.2.11 [**Artificial Intelligence**]: Distributed AI—*intelligent agents, languages, multiagent systems*

General Terms

Languages, Theory, Verification

Keywords

Agents, Model Checking, Verification, Programming

1. INTRODUCTION

In this paper, we present MABLE, a language intended for the design and automatic verification of multi-agent systems. MABLE is essentially a conventional imperative programming language, enriched by some additional constructs from the agent-oriented programming paradigm [13, 16, 19]. A MABLE system contains a number of agents, each of which is programmed using the MABLE imperative programming language. In addition, each agent in MABLE has 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. MABLE agents are able to communicate with one-another using `request` and `inform` performatives, in the style of the FIPA agent communication language [8]. 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 $\mathcal{LORA}$ [22], known as $\mathcal{MORA}$.

The MABLE language has been fully implemented. The implementation makes use of the SPIN system [11, 12], 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 [12]. 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. 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.

The remainder of this paper is structured as follows. We begin by introducing the theory of MABLE: its abstract syntax and formal semantics; the semantics are unusual in that we show how a semantics for the language can be given in terms of a belief-desire-intention logic. We then describe the MABLE language in detail — its syntax and informal semantics. We then describe how claims can be made about MABLE programs using $\mathcal{MORA}$, and how these claims can be automatically verified using MABLE. We then briefly describe the implementation of MABLE.

2. THEORETICAL FOUNDATIONS

MABLE is a language intended for the design and formal verification of multi-agent systems. Using MABLE, a designer can develop a multi-agent system using a language that combines constructs from conventional programming languages like C and JAVA with agent-specific features. In addition, a user can explicitly augment these programs with formal claims about the system behaviour, where these claims are expressed in a logic called $\mathcal{MORA}$, which combines elements of belief-desire-intention logics [17], temporal logics [14, 15], and dynamic logic [9]; by leveraging the model-

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

AAMAS'02, July 15-19, 2002, Bologna, Italy

checking capabilities of the SPIN system [11, 12], these claims can be automatically proved or disproved.

In this section, we focus on the theoretical foundations of MABLE. One of the key problems that MABLE is intended to address is that of *computational grounding*. Over the past two decades, many researchers have attempted to develop logical theories of rational agency. The theory of intention developed by Cohen and Levesque is perhaps best-known in this respect [4], and Rao and Georgeff's BDI logics have also received much attention [17, 22]; see [23] for a survey. These logics typically attempt to develop an axiomatic theory of the "mental state" of a rational agent. Traditionally, the tool of choice for such theories has been a multi-modal formalism, often combined with temporal and/or dynamic logic components. The semantics for such logics is usually given in terms of Kripke, or possible worlds, semantics [2]; the mathematics of correspondence theory for possible worlds semantics makes such an approach extremely attractive.

Unfortunately, most such logics tend to suffer from the problem of *computational grounding* [21]. Intuitively, this problem can be understood as follows. Suppose we have some axiomatic theory of agency, expressed in some logic $\mathcal{L}$, and we have some program π, implemented in some conventional programming language. Can we tell whether or not π implements the theory of agency? In general, we cannot. Technically, the problem is that conventional possible worlds semantics are not related in any way to the programs that we implement; they are instead a convenient but ultimately rather abstract mathematical representation. (Note that this is not the case for *epistemic logic* [7], which has a semantics, called interpreted systems, in which possible worlds are given a precise meaning in terms of the states of computer programs.)

A number of multi-agent programming languages have recently appeared, which attempt to bring logics of rational agency somewhat closer to programming languages (e.g., AGENT0 [19] and 3APL [10]). However, even for these languages, the link between programming language and logic is often not well-defined, and exception being GOLOG [13].

One of our aims with the MABLE language is to rectify this omission. MABLE is essentially a straightforward imperative programming language, enriched with some features from agent-oriented programming languages such as AGENT0 [19]. However, the semantics of MABLE are given in terms of a belief-desire-intention (BDI) logic called $\mathcal{LORA}$ [22]. That is, the effects of MABLE program constructs are defined with respect to the mental states of the agents that perform these actions. 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. The MABLE compiler (section 3.2) implements these semantics when it generates executable code; using the model checking component of MABLE, we can then automatically check BDI properties of MABLE systems.

2.1 LORA: A BDI Logic

The semantics of MABLE are defined using $\mathcal{LORA}$, a quantified, many-sorted multi-modal logic, which was defined in [22], and both draws upon and extends the work described in [4, 17]. $\mathcal{LORA}$ can be viewed as the well-known branching time logic CTL* [6], enriched by the addition of some further modal connectives for referring to the *beliefs*, *desires*, and *intentions* of agents [17], together with a simple apparatus for representing the actions performed by agents, which makes use of ideas from dynamic logic [9]. Note that, for simplicity, we assume that each agent has complete knowledge and that actions always succeed and are never delayed.

The logic is quantified and many-sorted; for simplicity, we do not allow functional terms in the language other than constants. Terms come in four sorts. First, we have terms that denote *agents*, and we use $i, j, \ldots$ and so on as variables ranging over agents. In addition, we have terms that denote (finite) *sets* of agents, i.e., groups — we use $g, g', \ldots$ as variables ranging over groups of agents. Next, we have terms that denote *sequences of actions* — we use $\alpha, \alpha', \ldots$ as terms denoting sequences of actions. Finally, we have terms that denote other objects in the environment. The logical apparatus of quantification is standard for quantified many-sorted logics.

The logic makes a distinction between formulae that express properties of states, and formulae that express properties of paths, or histories, through the temporal tree structure. The former are known as *state formulae*, the latter as *path formulae*. We begin our introduction by discussing the various *state formulae* operators (see Table 1 for an overview of the operators in the logic). First, we have a nullary operator true: a logical constant for truth. This formula will be satisfied wherever it is evaluated. Next, we have operators $(\mathsf{Bel}\ i\ \varphi)$, $(\mathsf{Des}\ i\ \varphi)$, and $(\mathsf{Int}\ i\ \varphi)$ which mean that agent i has a belief, desire, and intention of φ, respectively. An agent's beliefs intuitively correspond to the *information* that the agent has about its environment. For example, an agent might believe that the temperature of the room is 20 degrees Celsius, or that Bill Clinton is a liar. Agents can have *nested* beliefs; thus an agent might believe that Bill Clinton did not believe of himself that he was a liar. In this sense, MABLE represents a second-order intentional system. For technical reasons, we require that an agent only believes state formulae. Turning to desires, the idea is that an agent's desires represent those states of affairs that, ideally, it would like to bring about. For example, an agent might have a desire that the temperature in the room be 20 degrees Celsius, or might have a desire that Bill Clinton be impeached. As with beliefs, an agent's desires must be state formulae. An agent's intentions represent desires that the agent has committed to bring about; typically, intentions must be mutually consistent (whereas desires need not), and will persist over time [4]. In addition to these modal connectives, we have first-order equality: a formula $(\tau = \tau')$ will be true if τ and τ' denote the same individual. We allow state formulae to be combined using the usual connectives of classical logic: "$\neg$" ("not"), "$\vee$" ("or'), "$\wedge$" ("and"), "$\Rightarrow$" ("implies"), universal quantification, and so on.

We now consider path formulae. As we noted above, the idea is that path formulae express properties of a single path through a branching time structure. The main operator for expressing the properties of paths is "Happens". This operator takes a single argument: an *action expression*, and expresses the fact that this action expression is the first thing that happens on the path. Action expressions closely resemble the programs of dynamic logic, so the path formula $(\mathsf{Happens}\ \alpha)$ will be satisfied on some path if the action expression α is the first thing to occur on the path. Action expressions are formed using constructions that are well-known from dynamic logic: ";" (for sequential composition), "|" (for non-deterministic choice), "$*$" (for iteration), and "?" (for test actions). Thus the path formula $(\mathsf{Happens}\ \alpha; \alpha')$ means action α happens first on the path, and is immediately followed by α'. The formula $(\mathsf{Happens}\ \alpha|\alpha')$ means either α or α' happens first on the path. The formula $(\mathsf{Happens}\ \alpha*)$ means that the action α occurs one or more times at the start of the path. Finally, the formula $(\mathsf{Happens}\ \varphi?)$ means that the formula φ is satisfied in the first state of the path. Here, φ must be a state formula. To express the fact that some state of affairs is a necessary consequence of some action, we use the "Nec" operator. Thus $(\mathsf{Nec}\ \alpha\ \varphi)$ means that φ is a

Formula	Meaning
true	logical constant for truth
$(\mathsf{Bel}\ i\ \varphi)$	agent i believes φ
$(\mathsf{Des}\ i\ \varphi)$	agent i desires φ
$(\mathsf{Int}\ i\ \varphi)$	agent i has an intention of φ
$(\tau = \tau')$	τ is the same as τ'
$(i \in g)$	agent i is a member of group g
$(\mathsf{Happens}\ \alpha)$	action expression α happens next
$(\mathsf{Nec}\ \alpha\ \varphi)$	φ is a necessary consequence of α
$\mathsf{A}\varphi$	on all paths, φ holds (inevitably φ)
$\varphi\,\mathcal{U}\,\psi$	φ until ψ
$\Diamond\varphi$	sometime φ
$\Box\varphi$	always φ

Table 1: The main operators in $\mathcal{LORA}$

necessary consequence of action α; readers familiar with dynamic logic will recognise this as a $\mathcal{LORA}$ equivalent of $[\alpha]\varphi$ [9].

As with state formulae, compound path formulae can be made by combining path formulae using the connectives of classical logic. Path formulae in $\mathcal{LORA}$ can also be combined using the connectives of linear temporal logic (cf. [14, 15]): $\varphi\,\mathcal{U}\,\psi$ means φ is satisfied *until* ψ becomes satisfied; $\Diamond\varphi$ means φ is *eventually* satisfied; $\Box\varphi$ means φ is *always* satisfied.

State and path formulae are related to one another through *path quantifiers*, a concept borrowed from branching temporal logic [6]. The logic contains two such path quantifiers: "A", which means "on all paths", and "E", which means "on some path". These path quantifiers are unary modal connectives that are applied to path formulae to make state formulae. Thus $\mathsf{A}\varphi$ is a state formula, which will be satisfied in some state if the path formula φ is satisfied on all the paths through the temporal tree structure that originate from that state. The formula $\mathsf{E}\varphi$ is a state formula, which will be satisfied in some state if φ is satisfied on at least one path through the temporal tree structure that originates from that state.

2.2 Abstract Syntax

In this section, we present the abstract syntax of the MABLE agent programming language and define its semantics with respect to the BDI logic $\mathcal{LORA}$. By "abstract syntax", we mean that the *actual*, (concrete) syntax of MABLE is somewhat different — it includes a number of features that are not relevant from the point of view of programming language theory. For example, in the abstract syntax, we describe only one form of loop construct — the `while` loop. In the concrete syntax, however, there are several different kinds of loop (`for` loops, `do` loops, and so on). All these different types of loop can trivially be reduced to `while` loops, so it suffices to define the semantics of `while` loops only. In the same way, we have omitted other "syntactic sugar" features of the actual MABLE programming language as defined in section 3.

A MABLE system contains a finite number of MABLE *agents*. The basic form of a MABLE agent is

$$\texttt{agent}\ i\ \texttt{is}\ P$$

where i is the name of the agent and P is the program body. Each agent in a multi-agent system is assumed to have a unique name, drawn from a set $AgId = \{1, \ldots, n\}$ of *agent identifiers*. The main part of an agent, which determines its behaviour, is the program body P. The basis of program bodies in MABLE is a simple imperative language, containing iteration (`while` loops), sequence (the ";" constructor), selection (a form of the `if`, `then`, `else` statement), and assignment operations. Although readers will no doubt be well acquainted with such constructs, they take a novel form in MABLE, and some words of explanation are therefore necessary.

The most obvious difference between MABLE and conventional imperative languages is that MABLE has a `do` instruction, by which an agent i is allowed to execute any of a set $Ac = \{\alpha, \ldots\}$ of *external actions*. The simplest way to think of external actions is as *native methods* in a programming language like Java. They provide a way for agents to execute actions that do not simply affect the agent's internal state, but its external environment. The basic form of the `do` instruction is

$$\texttt{do}\ \alpha$$

where $\alpha \in Ac$ is the external action to be performed. When we incorporate communication into MABLE, we do so by modelling message sending as an external action to be performed.

MABLE programs have assignment operations of the form $x := e$, where x is a *program variable* and e is an (arithmetic) *expression*. As the syntax and semantics of such expressions are straightforward, we will not be concerned with defining them here; we will assume that the set *Exp* contains all syntactically acceptable expressions, which may be formed from the integers, a set of *program variables*, and the usual arithmetic operations (`+`, `-`, `*`, and `/` etc). We let *PVar* be the set of all program variables, and assume that agents do not share program variables.

MABLE contains a version of the regular `if`, `then`, `else` statement, allowing for the possibility of an agent being "uncertain" about something. The general format of the `if` statement is

$$\texttt{if}\ \varphi\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{unsure}\ P_3$$

The idea is that if the agent executing this instruction believes that the condition φ is true, then it will execute P_1 (where P_1 is a program). If it believes φ is false, then it will execute P_2. However, if it is *unsure* about whether φ is true or false, i.e., it neither believes φ nor $\neg\varphi$, then it will execute P_3. Note that we define the standard $\texttt{if}\ \varphi\ \texttt{then}\ P_1\ \texttt{else}\ P_2$ as $\texttt{if}\ \varphi\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{unsure}\ P_2$.

In a conventional programming language, conditions in `if` and `while` statements are only allowed to be dependent on program variables. Unusually, MABLE allows conditions in both `if` and `while` statements to be *arbitrary formulae of the BDI logic* $\mathcal{LORA}$ — any acceptable formula of $\mathcal{LORA}$ is allowed as a condition. To make this more concrete, consider the following:

$$\begin{array}{l}\texttt{if}\ (\mathsf{Bel}\ j\ x > 5)\ \texttt{then}\ x := x - 1\\ \quad\texttt{else}\ x := x + 1\ \texttt{unsure}\ x := 0\end{array}$$

The idea is that if the agent executing this instruction believes that agent j believes that $x > 5$, then the agent executing the instruction decrements the value of x by 1. If the agent executing the instruction believes it is *not* the case that agent j believes $x > 5$, then it increments the value of x by 1. Finally, if the agent executing the instruction has no opinion either way, it assigns the value 0 to x.

Notice the form of words used here: *the agent executing this loop instruction must believe that j believes x is greater than 5* — the condition does not depend on what j *actually* believes, but on what the agent executing the statement believes that j believes. As this example illustrates, conditions can thus refer to the mental state of other agents, in a similar way to the AGENT0 language [19].

The general form of a `while` construct, as in conventional programming languages, is

$$\texttt{while}\ \varphi\ \texttt{do}\ P$$

where φ is a condition and P is a program. As with `if` statements, conditions are not simply predicates over program variables, but are

$Prog$	$::=$	$\texttt{do}\ \alpha$	/* $\alpha \in Ac$ */
	$\mid$	$x := e$	/* $x \in PVar, e \in Exp$ */
	$\mid$	$\texttt{if}\ \varphi\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{unsure}\ P_3$	/* $\varphi \in wff(\mathcal{LORA}), P_i \in Prog$ */
	$\mid$	$\texttt{while}\ \varphi\ \texttt{do}\ P$	/* $\varphi \in wff(\mathcal{LORA}), P \in Prog$ */
	$\mid$	$P_1; P_2$	/* $P_i \in Prog$ */
$Agent$	$::=$	$\texttt{agent}\ i\ \texttt{is}\ P$	/* $i \in AgId, P \in Prog$ */
MAS	$::=$	$A_1 \| \cdots \| A_n$	/* $A_i \in Agent$ */

Figure 1: The abstract syntax of MABLE multi-agent systems.

arbitrary formulae of $\mathcal{LORA}$. Thus, for example, the following is an acceptable `while` loop in MABLE:

$$\texttt{while}\ (\mathsf{Bel}\ j\ x > 5)\ \texttt{do}\ x := x - 1$$

The idea is that, while the agent performing the `while` loop believes that agent *j* believes *x* is greater than 5, the agent will continually execute the statement $x := x - 1$.

Given a collection $A_1, \ldots, A_n$ of MABLE agents, they are composed into a multi-agent system by the parallel composition operator "$\|$": $A_1 \| \cdots \| A_n$. Note that, in this version of MABLE there is no mechanism for generating new agents.

Formally, the syntax of a MABLE multi-agent system is defined by the grammar in Figure 1.

2.3 Formal Semantics

We now give a formal semantics to MABLE. Our approach is inspired by the temporal semantics of concurrent programs [14, 15]. The idea behind temporal semantics is that the meaning of any program *P* is a set $[\![P]\!]$ of computations or runs, where each run represents one possible legal execution of the program *P*. Now models for temporal logic can be understood as sequences of states, which are in fact isomorphic to program runs. The semantics of a temporal logic formula is thus a set of such sequences — the sequences that satisfy the formula. The intuition in temporal semantics is that we can characterise the semantics of a program *P* as a temporal logic formula φ_P: the models of this formula correspond to the possible runs of the program *P*. A temporal semantics for programs is thus a function $[\![\ldots]\!]_{TL} : Prog \to wff(TL)$ which for every program $P \in Prog$ defines a temporal logic formula $[\![P]\!]_{TL}$, the models of which are exactly the possible runs of the program *P*.

In order to give a semantics to MABLE, we do not simply use temporal logic; instead, we use the BDI logic $\mathcal{LORA}$. The idea is that the semantics of a MABLE system will be defined as a formula of $\mathcal{LORA}$, which characterises the acceptable computations of the system, and the "mental state" of the agents in the system. Note that the formal semantics of our language is not complete: the main point is to show how a BDI semantics can be given to an imperative language. These semantics are then used in the compiler.

As we noted above, $\mathcal{LORA}$ extends branching time temporal logic with a rich collection of operators for describing both the beliefs, desires, and intentions of agents, and the actions of agents. Formally, the semantics of MABLE multi-agent systems are defined by three semantic functions, one each for multi-agent systems, agents, and agent program bodies:

$$\begin{array}{lcl} [\![\ldots]\!]_{MAS} & : & MAS \to wff(\mathcal{LORA}) \\ [\![\ldots]\!]_{Agent} & : & Agent \to wff(\mathcal{LORA}) \\ [\![\ldots]\!]_{Prog} & : & Prog \to wff(\mathcal{LORA}) \end{array}$$

In fact, the multi-agent system semantic function $[\![\ldots]\!]_{MAS}$ is defined in terms of the agent semantic function $[\![\ldots]\!]_{Agent}$, which is in turn defined in terms of the agent program semantic function $[\![\ldots]\!]_{Prog}$. The agent program semantic function is simply defined in terms of a function $[\![\ldots]\!]_{Exp} : Exp \to \mathbb{N}$, which gives the semantics of arithmetic expressions. This function will satisfy such properties as $[\![2+3]\!]_{Exp} = 5$ and so on. As the definition of this function is both standard and simple, we will not give it here; the interested reader is urged to consult, e.g., [20, pp55–61].

The three remaining semantic functions are defined in Figure 2. The idea is that the semantics are defined inductively by a set of definitions, one for each construct in the language.

The first rule defines the semantics of the `do` construct. The idea is that an agent which executes an instruction $\texttt{do}\ \alpha$ will intend that α happens next. The idea is closely related to that of *volitional commitment*, which states that an agent which intends to perform action α immediately does α [18, p442]. Our semantics states that all actions that are performed by an agent are actually intended by the agent; the name self used here refers to the agent that is executing the program. As we will see below, this name is used as a place-holder for the name of the agent that is executing the program body in all the semantic rules used to define program bodies.

The second rule gives the semantics of assignment statements. The idea is that an agent which executes a statement $x := e$ will subsequently believe that *x* has the value of the expression *e*, and, moreover, *x* really will have the value of *e*.

The third rule gives the semantics of the `if` construct. The idea is that given a statement $\texttt{if}\ \varphi\ \texttt{then}\ P_1 \texttt{else}\ P_2\ \texttt{unsure}\ P_3$, then if the agent believes φ, then P_1 will be executed; if it believes $\neg\varphi$, then P_2 will be executed. Now, using $\mathcal{LORA}$ allows us to capture a third possibility, where the agent neither believes φ nor $\neg\varphi$; this case is captured by the `unsure` part of the statement, in which case P_3 is executed.

The fourth rule gives the semantics of the `while` construct, and is rather conventional. The idea is to capture the semantics of `while` through its (least) *fixed point* characteristics: executing the statement $\texttt{while}\ \varphi\ \texttt{do}\ P$ is the same as executing the statement $\texttt{if}\ \varphi\ \texttt{then}\ (P; \texttt{while}\ \varphi\ \texttt{do}\ P)$.

Turning to the semantics of the semi-colon constructor, the idea is that $P_1; P_2$ will be executed if on all paths that originate from the current state, P_1 happens next and then P_2 happens. In the semantics, the meaning of P_1 is required to be satisfied, followed by the meaning of P_2.

As for the semantics of agent declarations, the idea is that a declaration $\texttt{agent}\ i\ \texttt{is}\ P$ binds a name *i* with the semantics of the program body *P*. We capture the semantics of this by taking the semantics $[\![P]\!]_{Prog}$ of the program *P* and systematically substituting *i* for the place-holder name self in $[\![P]\!]_{Prog}$.

Finally, we have a rule for the semantics of multi-agent systems: the semantics of a system $A_1 \| \cdots \| A_n$ is simply the conjunction of the semantics of the component agents A_i, together with some background assumptions ψ_{MAS}. The idea of the background assumptions is that these capture general properties of a multi-agent systems that

$$
\begin{array}{rcl}
[\![\texttt{do}\ \alpha]\!]_{Prog} & \hat{=} & (\mathsf{Int\ self\ A}(\mathsf{Happens}\ \alpha)) \\
[\![x := e]\!]_{Prog} & \hat{=} & \mathsf{A}\bigcirc((\mathsf{Bel\ self}\ (x = [\![e]\!]_{Exp})) \wedge (x = [\![e]\!]_{Exp})) \\
[\![\texttt{if}\ \varphi\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{unsure}\ P_3]\!]_{Prog} & \hat{=} & (\mathsf{Bel\ self}\ \varphi) \Rightarrow [\![P_1]\!]_{Prog} \\
 & & \wedge\ ((\mathsf{Bel\ self}\ \neg\varphi) \wedge \neg(\mathsf{Bel\ self}\ \varphi)) \Rightarrow [\![P_2]\!]_{Prog} \\
 & & \wedge\ (\neg(\mathsf{Bel\ self}\ \neg\varphi) \wedge \neg(\mathsf{Bel\ self}\ \varphi)) \Rightarrow [\![P_3]\!]_{Prog} \\
[\![\texttt{while}\ \varphi\ \texttt{do}\ P]\!]_{Prog} & \hat{=} & [\![\texttt{if}\ \varphi\ \texttt{then}\ (P; \texttt{while}\ \varphi\ \texttt{do}\ P)]\!]_{Prog} \\
[\![P_1; P_2]\!]_{Prog} & \hat{=} & \mathsf{A}(\mathsf{Happens}\ [\![P_1]\!]_{Prog}?; [\![P_2]\!]_{Prog}?) \\
[\![\texttt{agent}\ i\ \texttt{is}\ P]\!]_{Agent} & \hat{=} & [\![P]\!]_{Prog}[i/\mathsf{self}] \\
[\![A_1 \| \cdots \| A_n]\!]_{MAS} & \hat{=} & [\![A_1]\!]_{Agent} \wedge \cdots \wedge [\![A_n]\!]_{Agent} \wedge \psi_{MAS}
\end{array}
$$

Figure 2: The semantics of multi-agent systems: the functions $[\![\ldots]\!]_{Prog}$, $[\![\ldots]\!]_{Ag}$, $[\![\ldots]\!]_{MAS}$ are inductively defined in terms of the semantic function $[\![\ldots]\!]_{Exp}$ and some background assumptions represented by the $\mathcal{LORA}$ formula ψ_{MAS}.

are not captured directly by the semantics of the language.

Background Assumptions

First, we assume that if an agent i intends to immediately perform some action α, then this action is performed.

$$(\mathsf{Int}\ i\ \mathsf{A}(\mathsf{Happens}\ \alpha)) \Rightarrow \mathsf{A}(\mathsf{Happens}\ \alpha) \qquad (\psi^1_{MAS})$$

Next, we assume that if an agent i intends some action α, and a necessary consequence of α is φ, then i intends φ.

$$(\mathsf{Int}\ i\ \mathsf{A}(\mathsf{Happens}\ \alpha)) \wedge (\mathsf{Nec}\ \alpha\ \varphi) \Rightarrow (\mathsf{Int}\ i\ \varphi) \qquad (\psi^2_{MAS})$$

This is the "side effect" property [4, 17, 22]. It states that an agent intends the side effects of its intentions.

2.4 Introducing Communication

We now introduce communication actions: we allow an agent to perform two communicative acts, which we will refer to as `inform` and `request` respectively. The idea is that these are actions that an agent performs through the `do` instruction; a communicative action causes the corresponding message to be sent to the named recipient of the message. The abstract syntax of these communicative acts is defined by the following grammar:

$$
\begin{array}{lcll}
Ac & ::= & \texttt{inform}\ j\ \texttt{of}\ \varphi & /*\ j \in AgId, \varphi \in wff(\mathcal{LORA})\ */ \\
 & | & \texttt{request}\ j\ \texttt{to}\ \alpha & /*\ j \in AgId, \alpha \in Ac\ */
\end{array}
$$

Intuitively, when an agent i performs an action `inform` j `of` φ, the agent i is attempting to get agent j to believe the content of φ. When performing an action `request` j `to` α, the agent is attempting to get agent j to perform action α. However, as our agents are assumed to be autonomous, we do not assume that an agent will come to believe some statement simply because it has been `inform`ed of it, nor that an agent will carry out an action simply because it has been `request`ed to (see, e.g., [22, pp125–145] for a discussion). Instead, following the work of many researchers in the theory of speech acts (e.g., [5]), we assume that the effect of an `inform` message is to make the recipient j believe that the sender i intends that the recipient j believes φ; it is then up to j whether or not it actually comes to believe φ itself. Similarly, the effect of i `requesting` j to perform α is to make the recipient believe that the sender intends that the recipient intends to perform α. An obvious question is how the effects of a speech act are operationalised within an agent: how the receipt of a message by an agent causes this agent to change state. One possibility is that the virtual machine executing the agent programs is capable of performing this update; the details are not important for our purposes.

The following rules define the semantics of communication actions:

$$
\begin{array}{rcl}
[\![\texttt{do inform}\ j\ \texttt{of}\ \varphi]\!]_{Prog} & \hat{=} & (\mathsf{Int\ self\ A}(\mathsf{Happens}\ \mathit{inform}(\mathsf{self}, j, \varphi)) \\
[\![\texttt{do request}\ j\ \texttt{to}\ \alpha]\!]_{Prog} & \hat{=} & (\mathsf{Int\ self\ A}(\mathsf{Happens}\ \mathit{request}(\mathsf{self}, j, \alpha))
\end{array}
$$

In these definitions, $\mathit{inform}(\mathsf{self}, j, \varphi)$ is a $\mathcal{LORA}$ term that denotes the action of the agent denoted by self sending the agent denoted by j a message "$\mathit{inform}(\mathsf{self}, j, \varphi)$". (To make this concrete, imagine that a string "$\mathit{inform}(\mathsf{self}, j, \varphi)$" is sent from the agent denoted by self to j.) Similarly, $\mathit{request}(\mathsf{self}, j, \alpha)$ is a $\mathcal{LORA}$ term that denotes the action of the agent denoted by self sending the agent denoted by j a message "$\mathit{request}(\mathsf{self}, j, \alpha)$".

In order to give a semantics to communication, we must give some background assumptions ψ_{MAS} relating to communication. To do this, we must first make some choices, the most important of which is whether or not communication is assumed to be *guaranteed* or not. Another choice relates to whether or not messages are guaranteed to be delivered immediately, or simply delivered at some point in the future. This leads to the following possible background axioms for communication:

$$
\begin{array}{l}
(\mathsf{Nec}\ \mathit{inform}(\mathsf{self}, j, \varphi)\ \oplus\otimes(\mathsf{Bel}\ j\ (\mathsf{Int\ self}\ (\mathsf{Bel}\ j\ \varphi)))) \\
(\mathsf{Nec}\ \mathit{request}(\mathsf{self}, j, \alpha)\ \oplus\otimes(\mathsf{Bel}\ j\ (\mathsf{Int\ self}\ (\mathsf{Int}\ j\ \mathsf{A}\Diamond(\mathsf{Happens}\ \alpha)))))
\end{array}
$$

where:

- $\oplus$ is "A" (the universal path quantifier) if message delivery is guaranteed, and "E" (the existential path quantifier) otherwise; and
- $\otimes$ is "$\bigcirc$" ("next") if message delivery is immediate, and "$\Diamond$" ("at some time in the future") otherwise.

(Recall that the $\mathcal{LORA}$ formula $(\mathsf{Nec}\ \alpha\ \varphi)$ means that φ is a necessary consequence of the action φ: on all the paths where α is executed, φ follows [22, p82].)

3. THE MABLE IMPLEMENTATION

In this section, we describe the implemented MABLE language itself. Whereas in the preceding section, we described the abstract syntax and formal semantics of the language, in this section we describe the concrete syntax of the language — the syntax a programmer will actually use to write MABLE programs. The language has been implemented as a compiler, which translates MABLE systems into a form that can be processed by the SPIN model checker [11, 12]. The way in which the MABLE compiler fits in with the SPIN system is illustrated in Figure 3.

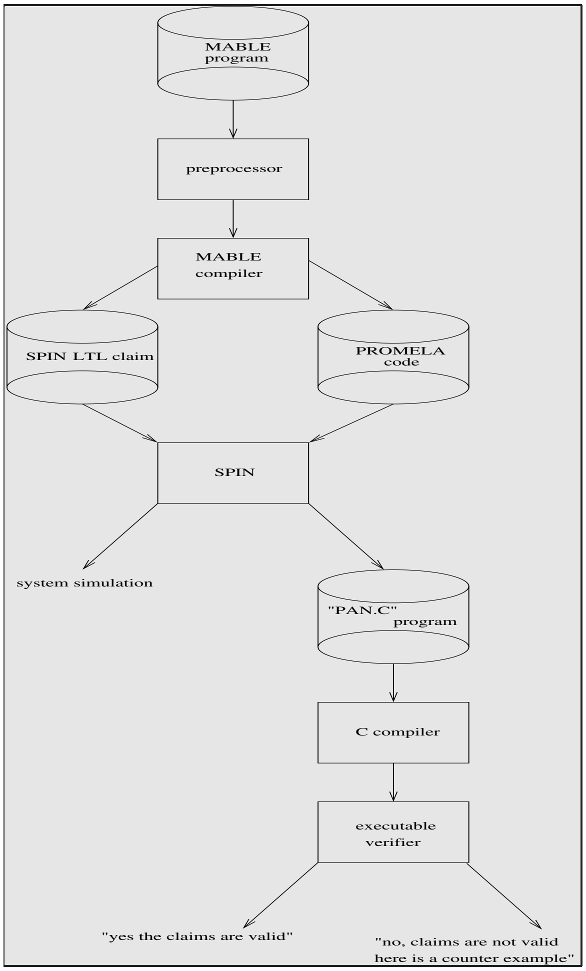

Figure 3: Operation of the MABLE system.

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. Shared variables are so called because they are implicitly part of a data structure shared by all agents. Shared variables are intended to allow a user to model shared resources in environments; for this reason, MABLE does not provide any built-in synchronisation for shared variables — if mutual exclusion is required over shared variables, then it is assumed that the agents will organise this themselves. 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; we say that all agent's 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 do so in order to discover their value*. They do this by executing a MABLE `observe` statement — this is a "sensor" action. When an agent performs such an action, 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.

In order to update 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` statements [3].

Output: The `print` command is the *only* output command available in MABLE. It causes its arguments to be sent to standard output (`stdout`): in most cases, this simply means the terminal from which the command was executed.

Conditions: Conditions in MABLE may be constructed from expressions with the usual relational operators (<, >, ==, ...). However, MABLE also permits conditions to contain *modalities*: in particular, belief, desire, and intention *modalities* — see Table 1. A BDI modality contains three components: the type of the modality (belief, desire, intention); the name of an agent; and a further condition. The intended meaning of the modality `(`*m ag c*`)` is that agent *ag* has attitude *m* towards the condition (predicate) *c*. The name *ag* must be the name of an agent in the system, and *c* must be a syntactically acceptable MABLE condition. As modalities are themselves conditions, modalities may be nested. For example, the following is a legal conditional expression in MABLE:

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

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.

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

Communication: As described above MABLE provides two built-in communication primitives: `inform` and `request`, inspired by the FIPA agent communication language [8]. The syntax of these is as follows:

`inform` *ag* `of` *c*`;`
`request` *ag* `to` *c*`;`

The effect of communication is to change the mental state of the recipient of the message, as described in section 2. By default, message delivery is guaranteed but asynchronous: thus when one agent sends another agent an `inform` message, the effect is to

```
fmla ::=
     forall IDEN ":" domain fmla     /* universal quantification */
  |  exists IDEN ":" domain fmla     /* existential quantification */
  |  any acceptable MABLE condition  /* primitive conditions */
  |  "(" fmla ")"                    /* parentheses */
  |  "[]" fmla                       /* always in the future */
  |  "<>" fmla                       /* sometime in the future */
  |  fmla "U" fmla                   /* until */
  |  "!" fmla                        /* negation */
  |  fmla "&&" fmla                  /* conjunction */
  |  fmla "||" fmla                  /* disjunction */
  |  fmla -> formula                 /* implication */

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

Figure 4: The syntax of $\mathcal{MORA}$ claims.

eventually make the recipient believe that the sender intends the recipient believe the message content.

Locks: In order to allow agents to synchronise their activities, MABLE provides a facility for *locking* critical sections of code. Essentially, 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 possess 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 code section; when it exits the code, the lock is released.

3.1 Claims and How to Verify Them

The most novel aspect of MABLE is that agent definitions may be interspersed with *claims* about the behaviour of agents, expressed in $\mathcal{MORA}$, a subset of the $\mathcal{LORA}$ language introduced in [22]. These claims can then be *automatically* checked; in this way, we can automatically verify the behaviour of MABLE systems. 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 4. The language of claims is thus that of quantified linear temporal BDI logic. 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. Below, we describe the way in which claims are dealt with during model checking

To illustrate the role of claims, consider the example in Figure 5. In this example, we have a shared variable, `a`, and two agents, `agent1` and `agent2`, whose behaviour is as follows.

- `agent1` counts up to 10 using variable `a`, and then informs `agent2` that `a == 10` (i.e., variable a has value 10). It then immediately starts counting to 15 using variable `a` again, and terminates.

- `agent2` waits until it believes that `agent1` intends that it believes that `a == 10` — this will be the case when `agent1` executes the `inform` statement. It then checks to see whether it believes that `a == 10`; if it does believe that `a == 10`, then it indicates that `agent1` is telling the truth, otherwise, it indicates that `agent2` is a liar (because `agent1` told `agent2` that `a` was 10 when it was not).

Notice that as `a` is a shared variable, `agent2` correctly knows its value without having to perform an `inform` instruction. Also, whether or not `agent2` will claim that `agent1` is a liar or not will actually depend on how the two agents are scheduled for execution: if, after sending the `inform` message to `agent2`, `agent1` starts incrementing `a` again before `agent2` gets to execute the `if` statement, then `agent2` will claim that `agent1` lies. In the current MABLE implementation, scheduling agents for execution is done randomly. Running the example with the compiler thus generates exactly this behaviour: on some runs, `agent2` claims that `agent1` lies, while on others, it claims that it tells the truth.

There is just one claim in this example, which when re-written $\mathcal{LORA}$ notation is:

$$\exists i \cdot \Diamond(\mathsf{Bel}\ i\ (\mathsf{Int}\ agent1\ (\mathsf{Bel}\ i\ a = 10)))$$

In other words, some agent i eventually comes to believe that $agent1$ intends that i believes a has the value 10. This claim is clearly valid, since `agent2` believes this. The MABLE compiler is capable of automatically translating this claim into the LTL form required by the SPIN model checker, and running SPIN to determine the truth or falsity of the claim. In this particular case, the claim can easily be verified on a standard desktop PC.

3.2 How the MABLE Compiler Works

In this section, we give a brief overview of the way in which the MABLE compiler works. There are three main components to the MABLE compiler: the way in which individual agents and their control constructs (e.g., loops) are implemented in PROMELA; the way in which belief-desire-intention states are implemented; and the way in which $\mathcal{MORA}$ claims are dealt with. The simplest of these is the implementation of basic control constructs. Although PROMELA is a relatively low-level language, it is straightforward to map MABLE's control constructs into those provided by PROMELA. Each agent in MABLE is implemented as a process (`proctype`) in PROMELA; additional PROMELA initialisation code is generated to automatically start agents simultaneously.

More interesting is the way that belief-desire-intention states are dealt with. The idea is to model these as finitely nested data structures (in the style of [1]). Predicates are represented by *propositional abstraction*: where a predicate appears in the context of a modality, it is automatically rewritten as a new proposition symbol. To implement the BDI semantics as described in section 2, we use *model annotation*. Thus when we generate the PROMELA code, we automatically annotate it with statements corresponding to the semantics. For example, when a message is received, the change in mental state on the part of the message recipient is automatically implemented by PROMELA code generated by the MABLE compiler.

Claims are dealt with using the following procedure:

- 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.

```
shared int a;

claim exists ag :  agent <>(believe ag (intend agent1 (believe ag a == 10)));

agent agent1 {
  for (a = 0; a != 10; a = a + 1)
    print("agent1:  a = %d \n", a);
  inform agent2 of (a == 10);
  for (a = 10; a != 15; a = a + 1)
  print("agent1:  a = %d \n", a);
}

agent agent2 {
  await (intend agent1 (believe agent2 a == 10));
  print ("agent 2 recvd mesg\n");
  if ((intend agent1 (believe agent2 a == 10)) && (a != 10))
    print("agent2:  agent1 is a LIAR!\n");
  else
    print("agent2:  agent1 tells the TRUTH!\n");
}
```

Figure 5: A final example.

- Predicates are removed by propositional abstraction: each predicate is replaced by proposition, the truth of which is bound to the predicate it replaces.

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.

4. CONCLUSIONS & FUTURE WORK

We have presented an imperative multi-agent programming language called MABLE, and a formal semantics for this language in terms of a BDI logic called $\mathcal{LORA}$. We have also described an implementation of this language, and described how claims about MABLE systems, expressed in a quantified linear temporal BDI logic called $\mathcal{MORA}$ (a cut-down version of $\mathcal{LORA}$), can be automatically checked by translating them into the form used by the SPIN model checking system. We are currently using and enhancing MABLE; we are using it in an EC project for verifying properties of electronic institutions, and in addition are enhancing its capabilities both at the programming language level (by providing more features for the programmer), and also by enhancing its model-checking features. We are also writing a MABLE to JAVA feature, which will automatically generate JAVA code from MABLE systems. Finally, our future work in this area also concerns exploring efficiency and completeness issues.

5. REFERENCES

[1] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multi-agent systems. *Journal of Logic and Computation*, 8(3):401–424, 1998.

[2] B. Chellas. *Modal Logic: An Introduction*. Cambridge University Press: Cambridge, England, 1980.

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

[4] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[5] P. R. Cohen and H. J. Levesque. Rational interaction as the basis for communication. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication*, pages 221–256. The MIT Press: Cambridge, MA, 1990.

[6] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[7] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.

[8] The Foundation for Intelligent Physical Agents. See `http://www.fipa.org/`.

[9] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press: Cambridge, MA, 2000.

[10] 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.

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

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

[13] 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 Vol. 1037)*, pages 331–346. Springer-Verlag: 1996.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag: Berlin, Germany, 1992.

[15] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer-Verlag: Berlin, Germany, 1995.

[16] 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: Proc. Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Vol. 1038)*, pages 42–55. Springer-Verlag: 1996.

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

[18] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proc. Knowledge Representation and Reasoning*, pages 439–449, 1992.

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

[20] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press: Cambridge, MA, 1993.

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

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

[23] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.