Model Checking AgentSpeak

Rafael H. Bordini Michael Fisher Carmen Pardavila Michael Wooldridge
Department of Computer Science, University of Liverpool,
Liverpool L69 7ZF, U.K.
{R.Bordini, M.Fisher, C.Pardavila, M.J.Wooldridge}@csc.liv.ac.uk

ABSTRACT

This paper introduces AgentSpeak(F), a variation of the BDI logic programming language AgentSpeak(L) intended to permit the model-theoretic verification of multi-agent systems. After briefly introducing AgentSpeak(F) and discussing its relationship to AgentSpeak(L), we show how AgentSpeak(F) programs can be transformed into Promela, the model specification language for the Spin model-checking system. We also describe how specifications written in a simplified form of BDI logic can be transformed into Spin-format linear temporal logic formulæ. With our approach, it is thus possible to automatically verify whether or not multi-agent systems implemented in AgentSpeak(F) satisfy specifications expressed as BDI logic formulæ. We illustrate our approach with a short case study, in which we show how BDI properties of a simulated auction system implemented in AgentSpeak(F) were verified.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Languages and structures, Multiagent systems*; D.2.4 [**Software/Program Verification**]: Model checking

General Terms

Languages, Verification

Keywords

Model Checking, BDI Logic Programming, AgentSpeak, Spin

1. INTRODUCTION

As multi-agent systems come to the attention of a wider technical community, there is an ever increasing requirement for tools supporting the design and implementation of such systems. While these tools should be usable by a general computing audience, there should also be strong theoretical foundations for such tools, so that formal methods can be used in the design and implementation processes. In particular, the *verification* of multi-agent systems — showing that a system is correct with respect to its stated requirements — is an increasingly important issue, especially as agent systems start to be applied to safety-critical applications such as autonomous spacecraft control [12, 7].

Currently, the most successful approach to the verification of computer systems against formally expressed requirements is that of *model checking* [4]. Model checking is a technique that was originally developed for verifying that finite state concurrent systems implement specifications expressed in temporal logic. Although model checking techniques have been most widely applied to the verification of hardware systems, they have increasingly been used in the verification of software systems and protocols [9].

Our aim in this paper is to present model checking techniques for verifying systems implemented in AgentSpeak(L). The AgentSpeak(L) BDI logic programming language was created by Rao [13], and was later developed into a more practical programming framework [2]. While the theoretical foundations of AgentSpeak(L) are increasingly well understood [3], there has not been, to the best of our knowledge, any significant work on verifying systems implemented in AgentSpeak(L), or any other BDI logic programming language for that matter.

We begin by introducing AgentSpeak(F), a variation of AgentSpeak(L) intended to permit its algorithmic verification. Next, we show how AgentSpeak(F) programs can be automatically transformed into Promela, the model specification language for the Spin model-checking system [9]. We also present a simplified form of BDI logic in which specifications can be written. The language is defined so as to make it possible to transform those specifications into Spin-format linear temporal logic formulæ. In this way, we can verify automatically whether or not multi-agent systems implemented in AgentSpeak(F) satisfy specifications expressed as BDI logic formulæ. We illustrate our approach with a brief case study of a simplified simulation of an abstract auction system with three bidding agents. We give the AgentSpeak(F) code for those three agents, and show three BDI specifications that are satisfied by the auction system as verified by the Spin model checker.

The paper is structured as follows. Section 2 provides an overview of AgentSpeak(L), while we introduce AgentSpeak(F) in Section 3. Section 4 contains the main contribution of this paper: it shows how AgentSpeak(F) can be transformed into Promela (Section 4.1), and describes the BDI logical languages used for specifications as well as how the BDI modalities are interpreted in terms of AgentSpeak(L) data structures (Section 4.2). We then present our auction system case study in Section 5, give related work in Section 6, and conclude, in Section 7, with discussion and future work.

2. AgentSpeak(L)

In [13], Rao introduced the AgentSpeak(L) programming language. It is a natural extension of logic programming for the BDI

AAMAS'03, July 14–18, 2003, Melbourne, Australia.

agent architecture, and provides an elegant abstract framework for programming BDI agents. In this paper, we only give a very brief introduction to AgentSpeak(L); see [13, 3] for more details.

An AgentSpeak(L) agent is created by the specification of a set of base beliefs and a set of plans. A *belief atom* is simply a first-order predicate in the usual notation, and belief atoms or their negations are *belief literals*. An *initial set of beliefs* is just a collection of ground belief atoms.

AgentSpeak(L) distinguishes two types of goals: *achievement goals* and *test goals*. Achievement goals are predicates (as for beliefs) prefixed with the '`!`' operator, while test goals are prefixed with the '?'operator. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, these start off the execution of *subplans*.) A *test goal* states that the agent wants to test whether the associated predicate is a belief (i.e., whether it can be unified with that agent's base beliefs).

Next, the notion of a *triggering event* is introduced. It is a very important concept in this language, as triggering events define which events may initiate the execution of plans; the idea of *event*, both internal and external, will be made clear below. There are two types of triggering events: those related to the *addition* ('`+`') and *deletion* ('`-`') of mental attitudes (beliefs or goals).

Plans refer to the *basic actions* that an agent is able to perform on its environment. Such actions are also defined as first-order predicates, but with special predicate symbols (called *action symbols*) used to distinguish them. The actual syntax of AgentSpeak(L) programs is based on the definition of plans, below. Recall that the designer of an AgentSpeak(L) agent specifies a set of beliefs and a set of plans only.

If e is a triggering event, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n$ are goals or actions, then e `:` b_1 `&` ... `&` b_m `<-` h_1 `;` ... `;` h_n`.` is a *plan*. An AgentSpeak(L) plan has a *head* (the expression to the left of the arrow), which is formed from a triggering event (denoting the purpose for that plan), and a conjunction of belief literals representing a *context* (separated from the triggering event by ':'). The conjunction of literals in the context must be satisfied if the plan is to be executed (the context must be a logical consequence of that agent's current beliefs). A plan also has a *body*, which is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan is triggered.

Although the interpretation of AgentSpeak(L) programs is not explained here, some of the related notions are given next. *Intentions* are particular courses of actions to which an agent has committed in order to achieve a particular goal: each intention is a stack of *partially instantiated plans*, i.e., plans where some of the variables have been instantiated. An *event*, which may trigger the execution of a plan, can be *external*, when originating from perception of the agent's environment, or *internal*, when generated from the agent's own execution of a plan (e.g., an achievement goal within a plan body is a goal-addition event which may be a triggering event). Formally, an event is a pair $\langle te, i\rangle$, where te is a triggering event and i is an intention. For internal events, i is the intention which generated the event, and for external events i is T (the *true intention*).

An AgentSpeak(L) *agent* is formally defined by a tuple $\langle E, B, P, I, A, \mathcal{S}_\mathcal{E}, \mathcal{S}_\mathcal{O}, \mathcal{S}_\mathcal{I}\rangle$, where E is a set of events, B is a set of base beliefs, P is a set of plans, I is a set of intentions, and A is a set of actions (where the actions the agent decides to execute are inserted). The selection function $\mathcal{S}_\mathcal{E}$ selects an event from the set E; the selection function $\mathcal{S}_\mathcal{O}$ selects an option or an applicable plan from a set of applicable plans; and $\mathcal{S}_\mathcal{I}$ selects an intention from the set I (the chosen intention is then executed).

3. AgentSpeak(F)

Recall that our main goal in this research is to facilitate model checking of AgentSpeak(L) systems. But model checking as a paradigm is predominantly applied to *finite state* systems. A first key step in our research was thus to restrict AgentSpeak(L) to finite state systems: the result is AgentSpeak(F), a finite state version of AgentSpeak(L).

In order to ensure that systems to be model-checked are finite state, the maximum size of types, data structures, and communication channels must be specified. This means that, for a translator from AgentSpeak(L)-like programs into a model checking system to work, a series of parameters stating the expected maximum number of occurrences of certain AgentSpeak(L) constructs need to be given. The list below describes all the parameters needed by our automatic translator.

M_{Term}: maximum number of *terms* in a predicate or an action (i.e., the maximum arity for a predicate or action symbol);

M_{Conj}: maximum number of *conjuncts* (literals) in a belief forming a plan's context;

M_{Var}: maximum number of different *variables* in a plan;

M_{Inst}: maximum number of *instances* (entries) in the belief base of the same predicate symbol at a time;

M_{Bel}: maximum number of *beliefs* an agent can have at any moment in time in its belief base;

M_{Ev}: maximum number of pending *events*, i.e., the maximum number entries in the event queue that an agent will store at a time; this should be set by considering how dynamic the environment is expected to be;

M_{Int}: maximum number of *intended means* at a time; that is, the number of different instances of plans rather than the number of stacks of plans in the set of intentions;

M_{Act}: maximum number of *actions* requested by the agents that may have to wait for the environment to handle;

M_{Msg}: maximum number of *messages* (generated by inter-agent communication) that an agent can store at a time.

Note that the first three parameters (M_{Term}, M_{Conj}, and M_{Var}) are inputs to the automatic translator, but they could be determined purely by syntactic processing. The others are restrictions on the data structures used in an AgentSpeak(L) interpreter, as explained in Section 4.1. Some of these parameters will be used in the syntax of AgentSpeak(F), as seen below.

The grammar in Figure 1 gives the syntax of AgentSpeak(F). In that grammar, P stands for any predicate symbol and A for any action symbol. Terms t_i associated with them are either constants or variables. As in Prolog, an uppercase initial letter is used for variables and lowercase for constants and predicate symbols (c.f., Prolog atoms). Note that first order terms (c.f., Prolog structures) are not allowed in the present version of AgentSpeak(F); the next section discusses this restriction further.

There are some special action symbols which are denoted by an initial '.' character (they are referred to as internal actions in [2]). The action '.send' is used for inter-agent communication, and is interpreted as follows. If an AgentSpeak(F) agent l_1 executes .send(l_2, ilf, at), a message will be immediately inserted in the mailbox of agent l_2, having l_1 as sender, illocutionary force ilf, and propositional content at (an atomic AgentSpeak(F) formula). At this stage, only three illocutionary forces can be used: tell, untell, and achieve (unless others are defined by the user). They have the same informal semantics as in the well-known KQML agent

$$
\begin{array}{lcll}
ag & ::= & bs \quad ps & \\
bs & ::= & at_1\textbf{.} \quad \ldots \quad at_n\textbf{.} & (0 \le n \le M_{Bel}) \\
at & ::= & \mathsf{P}(t_1\textbf{,} \ldots \textbf{,} t_n) & (0 \le n \le M_{Term}) \\
ps & ::= & p_1 \quad \ldots \quad p_n & (n \ge 1) \\
p & ::= & te \textbf{ : } ct \textbf{ <- } h \textbf{ .} & \\
te & ::= & \textbf{+}at \quad | \quad \textbf{-}at & \\
 & | & \textbf{+}g \quad | \quad \textbf{-}g & \\
ct & ::= & at \quad | \quad \texttt{true} & \\
 & | & \texttt{not (} at \texttt{)} & \\
 & | & ct_1 \textbf{ \& } \ldots \textbf{ \& } ct_n & (1 \le n \le M_{Conj}) \\
h & ::= & \mathsf{A}(t_1\textbf{,} \ldots \textbf{,} t_n) & (0 \le n \le M_{Term}) \\
 & | & g \quad | \quad u \quad | \quad h \textbf{ ; } h & \\
g & ::= & \textbf{!}at \quad | \quad \textbf{?}at & \\
u & ::= & \textbf{+}at \quad | \quad \textbf{-}at &
\end{array}
$$

Figure 1: The Syntax of AgentSpeak(F)

communication language. In particular, achieve corresponds to including at as a goal addition in the receiving agent's set of events; tell and untell change the belief base and the appropriate events are generated. These communicative acts only change an agent's internal data structures after user-defined trust functions are checked. There is one specific trust function for belief changes, and another for achievement goals. The latter defines a power relation (as other agents have power over an agent's goals), whereas the belief trust function simply defines the trustworthiness of information sources.

Another internal action symbol that is available is .print, which takes a string as parameter and is used for agents to print out messages. Other pre-defined internal actions are, for example, used for conditional operators and arithmetic operations.

Syntactically, the main difference between AgentSpeak(F) and AgentSpeak(L) is that first order terms are not allowed, and there are the informed limits on the number of beliefs, terms, and conjuncts indicated by the use of M_{Bel}, M_{Term}, and M_{Conj} above. There is also the limit on the number of variables in a plan (M_{Var}) which was not made explicit in the grammar. Note, however, that M_{Bel} is the maximum number of beliefs in the belief base at any moment during the agent's execution, not just the maximum number of initial beliefs.

We now consider other restrictions of AgentSpeak(F) in relation to AgentSpeak(L). Our current implementation imposes some restrictions on certain features of AgentSpeak(L). In particular, it is presently not possible to use: (i) uninstantiated variables in triggering events; (ii) uninstantiated variables in negated literals in a plan's context (as originally defined by Rao [13]); (iii) the same predicate symbol with different arities (at present, the different predicates would be treated as the same, with either random extra arguments or ignoring some of them); (iv) first order terms (rather than just constants and variables).

The first restriction means that an achievement goal cannot be called with an uninstantiated variable (a usual means for a goal to return values to be used in the plan where it was called). However, as mentioned in [10], this restriction can be overcome by storing such values in the belief base, and using test goals to retrieve them. Hence, syntactic mechanisms for dealing with this restriction can be implemented (i.e., this problem can be solved by preprocessing).

As for the second restriction, the interpreter presented in [2] allows for uninstantiated variables in negated literals. However, this was not allowed in Rao's original definition of AgentSpeak(L), as it complicates slightly the process of checking a plan's context. Thus, the second restriction is not an unreasonable one.

4. MODEL CHECKING AgentSpeak(F)

We now move on to the key contribution of this paper: we describe how AgentSpeak(F) programs can be translated into Promela, the model specification language for the Spin model checker. We then describe the logical language used to specify BDI properties of multi-agent systems written in AgentSpeak(F). Throughout this section, we presuppose some familiarity with Promela [9], as space restrictions prevent a detailed account here.

4.1 Promela Model of AgentSpeak(F)

A summary of the Promela model of an AgentSpeak(F) interpreter (i.e., for one agent) is shown in Figure 2. Each identifier used in the AgentSpeak(F) source code (i.e., identifiers for predicate and action symbols and for constants) is defined in Promela as a macro for an integer number which represents that symbol uniquely. This is necessary because Promela cannot handle strings. AgentSpeak(F) variables are declared as integer Promela variables[1].

Data Structures

A number of Promela *channels* are used to handle most of the data structures needed by an AgentSpeak(L) interpreter; the use of Promela channels as lists had already been pointed out in [7]. All such channels are described below.

Channel ***b*** represents the agent's *belief base*. The type for the massages stored in this channel is composed of $M_{Terms} + 1$ integers (one to store the predicate symbol and at most M_{Terms} terms). The ***b*** channel is declared to store at most M_{Bel} messages. A similar channel called ***p*** stores the percepts. This is changed by the environment and read by all agents for belief revision. The format and number of messages is as for the ***b*** channel. Channel ***m*** is used for inter-agent communication. Messages in it contain the identification of the sender agent, the illocutionary force associated with the communication, and a predicate (as for beliefs). It is bounded to at most M_{Msg} messages.

Before we go on describing the channels used as data structures, we need to explain how intentions are handled. The bodies of plans are translated into Promela inline procedures. These are called whenever the interpreter requires an intended plan instance to run its next formula. The data about each intended means is stored in an array called ***i_data***. Accordingly, intended means can be identified by an index to an entry in this data structure. In fact, an AgentSpeak(L) intention is represented here by the index to the entry in ***i_data*** that is associated the plan on top of it; this is explained in detail later on.

Next, a channel called ***e*** (of size M_{Ev}) is used to store *events*. The message type here is formed by: (i) an integer to store an index to ***i_data*** (representing an AgentSpeak(L) intention[2]); (ii) a boolean defining whether the event is an addition or deletion; (iii) another boolean defining whether the event is (an addition or deletion of) a belief or a goal; and (iv) $M_{Terms} + 1$ integers to store a predicate as before.

Channel ***i***, used for scheduling *intentions*, stores messages of one integer, as only indices (to ***i_data***) of plan instances that are enabled for execution need to be stored. This corresponds to the plans on top of each of the stacks of plans in an agent's set of intentions. Both ***i*** and ***i_data*** have size M_{Int}. Given that we are using by default a "round-robin" intention selection function (as in [10]),

[1]Name clash is avoided by having internal variables (i.e., the ones needed by the AgentSpeak(F) interpreter code in Promela) being prefixed with '_', which is not a valid initial character for AgentSpeak(L) identifiers.

[2]Recall that an AgentSpeak(L) event is a tuple $\langle te, i \rangle$ where i is the intention that generated the triggering event te.

plan instances that are ready to be scheduled insert their indices (to ***i_data***) at the end of ***i***. The first index in channel ***i*** specifies the next plan that will have a given formula in its body chosen for execution. More on intention selection is mentioned in the next section.

Finally, the ***a*** channel (for *actions*) stores at most M_{Act} messages of the same type as ***b*** plus an identification of the agent requesting the action. Recall that an action has the same format as a belief atom (the difference in practice is that they appear in the body of plans).

The whole multi-agent system code in Promela will have arrays of the channels described above, one for each agent in the system. Only channels ***p*** and ***a*** are unique. They work as connection points with the environment, which is accessed by all agents. The environment is implemented as a Promela process type called `Environment`, which is defined by the user. It reads actions from channel ***a*** (which is written into by all agents) and changes the percepts that are stored in channel ***p*** (which is read by all agents).

The Interpretation Cycle

The AgentSpeak(L) interpretation cycle is summarised in Figure 2 (it shows the structure of the code generated for one of the agents). When an interpretation cycle starts, the agent checks its "mail box", and processes the first message in channel ***m***. The effects of the illocutionary forces that can be used, as mentioned in Section 3, are defined in an inline procedure `CheckMail` in a header file. This can be altered by the user to change or extend the semantics of communication acts, if necessary. Note that checking for messages is not explicitly mentioned in the original definitions of the abstract interpreter for AgentSpeak(L) [13, 5]. We here have separate stages in the interpretation cycle for considering inter-agent communication and perception of the environment, then belief revision takes care of both sources of information (in the figure, perception of the environment is implicit within belief revision). The trust functions (mentioned in Section 3) associated with this belief revision process are read from a header file. Unless the inline procedures `TrustTell` and `TrustAchieve` are redefined by the user, full trust is assumed among agents.

Next, the agent runs its belief revision function ("BRF" in the figure). The one used here is a simple piece of code composed of two Promela `do` loops. The first one checks all percepts (in ***p***) and adds to the belief base (channel ***b***) all those that are not presently there. This generates corresponding belief-addition events (of format $\langle \textbf{+}at, \mathsf{T}\rangle$). The second loop checks for current beliefs that are no longer in the percepts, and removes them. This generates the appropriate belief-deletion events (i.e., $\langle \textbf{-}at, \mathsf{T}\rangle$). It is, of course, a comparatively simple belief revision function, but quite appropriate for ordinary AgentSpeak(L) programs. The belief revision function is in a header file generated by the translator, and may be changed by the user if a more elaborate function is required.

Then, an event to be handled in this interpretation cycle has to be chosen. Events are handled via a FIFO policy here. Thus, when new events are generated, they are inserted in the end of ***e***, and the first message in that channel is selected as the event to be handled in the current cycle. The heads of all plans in an agent's plan library are translated into a sequence of attempts to find a relevant and applicable plan. Each such attempt is implemented by a matching of the triggering event against the first event in ***e***, and checking whether the context is a logical consequence of the beliefs. This is implemented as nested loops based on M_{Conj} auxiliary channels of size M_{Inst}, storing the relevant predicates from the belief base; the loops go on until a unification is found (or none is possible).

If the attempt for a plan p_j is successful, then it is considered as the intended means for the selected event. (Note that the $\mathcal{S}_{\mathcal{O}}$ selection function is implicitly defined as the order in which plans are written in the code.) At this point, a free space in ***i_data***, the array storing intention data, is needed (see `FindFreeSpace` in the figure). This space is initialised with the data of that intended means stating that: it is an instance of plan p_j; the formula in the body of the plan to be executed next is the first one (by initialising a *formula counter*); the triggering event[3] with which this plan is associated; the index in this array of the intention which generated the present event (if it was an internal one); and the binding of variables[4] for that plan instance.

There are some issues that have to be considered in relation to event selection and the creation of new intentions. The first message in channel ***e*** is always removed. This means that the event is discarded if no applicable plan was found for it. Also, recall that the user defines M_{Int}, specifying the maximum expected number of intentions an agent will have at any given time. Recall that this corresponds to the maximum number of plan instances in an agent's set of intentions (not the number of stacks of plans allowed in it). If any agent requires more than M_{Int} intended means, a Promela assertion will fail, interrupting the verification process (and similarly for the other translation parameters).

Finally, channel ***i*** is used for scheduling the execution of the various intentions an agent may have. As a round-robin like scheduler is assumed by default, using a channel for this is quite straightforward. Indices of the ***i_data*** array currently in ***i*** are used as a reference for the present intended means. When an intended means is enabled for execution, its index is sent to channel ***i***. The integer value idx in the first message in that channel is used as an index to access the intention data that is necessary for executing its next formula. This is done by calling an inline procedure according to the plan type stated in ***i_data***$[idx]$ (and idx is sent as a parameter to that inline procedure).

Plan bodies given in AgentSpeak(L) are translated into Promela inline procedures. Whenever these procedures are called, they only run the code that corresponds to the next formula to be executed (by checking the formula counter in the intention data). After executing the code for the current formula, the formula counter is incremented. Then the index in ***i_data*** for this intended means (idx received as parameter) is inserted again in channel ***i***, meaning that it is ready to be scheduled again. However, this is not done when the corresponding formula was an achievement goal; this is explained further below. When the last formula is executed, idx is no longer sent to ***i***, and the space in ***i_data*** for that plan instance is freed.

The translation of each type of formula that can appear in a plan body is relatively simple. Basic actions are simply appended to the ***a*** channel, with the added information of which agent is requesting it. The user-defined environment should take care of the execution of the action. Addition and deletion of beliefs is simply translated as adding or removing messages to/from the ***b*** channel, and including the appropriate events in ***e***. Test goals are simply an attempt to match the associated predicate with any message from channel ***b***. The results in the Promela variables representing uninstantiated variables in the test goal are then stored in ***i_data***, so that these values can be retrieved when necessary in processing subsequent formulæ. Achievement goals, however, work in a slightly different way from other types of formula.

When an achievement goal appears in the body of a plan in a running intention, all that happens is the generation of the appropriate internal event. Suppose the index in ***i_data*** of the plan instance on top of that intention is i_1. The intention that generated the event

[3]This is needed for retrieving information on the desired and intended formulæ of an agent.

[4]This is stored in an array of size M_{Var}.

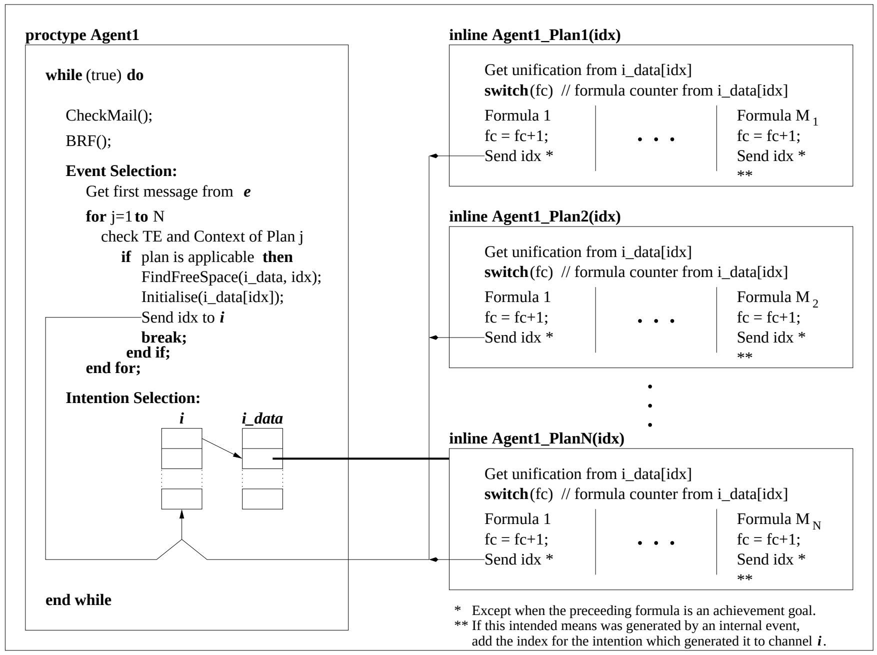

Figure 2: Abstract Promela Model for an AgentSpeak(F) Agent

is *suspended* until that event is selected in a reasoning cycle. In the Promela model, this means that we have to send a message to channel ***e***, but the formula counter is not incremented, and index i_1 is not sent to ***i***. This means that the plan instance in i_1 is not enabled for scheduling. However, the generated event will have i_1 to mark the intention that generated it. When an intended means is created for that event, i_1 will be annotated in ***i_data*** as the index of the intention that created it. All inline procedures generated as translation of plan bodies check, after the last formula is selected to run, whether there is an intention index associated with the entry in ***i_data*** they receive as parameter. If there is, that index should now be sent to ***i***, thus allowing the previously suspended intended means to be scheduled again.

This completes a reasoning cycle (a cycle of interpretation of an AgentSpeak(L) program). Each of the four main parts in the cycle (as seen in Figure 2), namely belief revision, checking inter-agent communication, event selection (and generating a new intended means for it), and intention selection (and executing one formula of it), are atomic steps in Promela. This means that, during model checking, Spin will consider all possible interleavings of such atomic operations being executed by all agents in the multi-agent system. This captures the different possible execution speeds of the agents.

The event selection and intention selection parts of the interpretation cycle always use the first messages in channels ***e*** and ***i***, respectively. However, before those parts of the cycle, two inline procedures are called. These procedures, named `SelectEvent` and `SelectIntention` have no effect by default, so channel ***e*** is used as a queue of events, and ***i*** provides a round-robin scheduler. Users can have control over event and intention selection by including code in the definition of those procedures. Such code would change the order of the messages in ***e*** or ***i*** (in particular the first ones) thus determining the event or intention that is going to be selected next.

The whole multi-agent system is started up in the Promela `init` process. It runs the user-defined `Environment` process and then creates one process for each agent defined in the translation process. The next section discusses the way in which BDI properties are specified for model-checking an AgentSpeak(F) multi-agent system created in the way described so far.

4.2 Verifying BDI Properties

Ideally, we would like to be able to verify that systems implemented in AgentSpeak(L) satisfy (or do not satisfy) properties expressed in BDI logic. In this section, we show how BDI logic properties can be mapped down into Spin-format Linear Temporal Logic (LTL) formulæ and associated predicates over the data structures in the Promela system.

In [3], a way of interpreting the informational, motivational, and deliberative modalities of BDI logics for AgentSpeak(L) agents was given. This was used to prove which of the asymmetry thesis principles [15] are enforced by AgentSpeak(L). In this work, we use that same framework for interpreting the B-D-I modalities in terms of data structures within the Promela model of an AgentSpeak(F) agent in order to translate (temporal) BDI properties into Promela never-claims. The particular logical language that is used for specifying such properties is given towards the end of the section.

The definitions in [3] are based on the operational semantics of AgentSpeak(L) [11]. The configurations of transition system giving such operational semantics are defined as a pair $\langle ag, C\rangle$, where an agent $ag = \langle bs, ps\rangle$ is defined as a set of beliefs bs and a set of plans ps (see Section 3), and C is the agent's present circumstance defined as a tuple $\langle I, E, A, R, Ap, \iota, \rho, \varepsilon\rangle$, where I, E, and A are as given in the definition of an AgentSpeak(L) agent in Section 2 (the others are not relevant here).

We here give only the main definitions within [3]; the argumentation on the proposed interpretation is omitted. In particular, discussion is given in that paper on the interpretation of intentions and desires, as the *belief* modality is clearly defined in AgentSpeak(L). We say that an AgentSpeak(L) agent ag, regardless of its circumstance C, believes a formula φ iff it is included in the agent's belief base; that is, for an agent $ag = \langle bs, ps \rangle$:

$$\text{BEL}_{\langle ag,C\rangle}(\varphi) \equiv \varphi \in bs.$$

Note that a closed world is assumed, so $\text{BEL}_{\langle ag,C\rangle}(\varphi)$ is true if φ is included in the agent's belief base, and $\text{BEL}_{\langle ag,C\rangle}(\neg\varphi)$ is true otherwise, where φ is an atom (i.e., at in Section 3).

Before giving the formal definition for the *intention* modality, we first define an auxiliary function $agls : \mathcal{I} \to \mathbb{P}(\Phi)$, where $\mathcal{I}$ is the domain of all individual intentions and Φ is the domain of all atomic formulæ (as mentioned above). Recall that an intention is a stack of partially instantiated plans, so the definition of $\mathcal{I}$ is as follows. The empty intention (or true intention) is denoted by T, and $\mathsf{T} \in \mathcal{I}$. If p is a plan and $i \in \mathcal{I}$, then also $i[p] \in \mathcal{I}$. The notation $i[p]$ is used to denote the intention that has plan p on top of another intention i, and C_E denotes the E component of C (and similarly for the other components). The $agls$ function below takes an intention and returns all achievement goals in the triggering event part of the plans in it:

$$\begin{array}{lcl} agls(\mathsf{T}) & = & \{\} \\ agls(i[p]) & = & \begin{cases} \{at\} \cup agls(i) & \text{if } p = \texttt{+!}at \texttt{ : } ct \texttt{ <- } h\texttt{.} \\ agls(i) & \text{otherwise.} \end{cases} \end{array}$$

Formally, we say an AgentSpeak(L) agent ag intends φ in circumstance C if, and only if, it has φ as an achievement goal that currently appears in its set of intentions C_I, or φ is an achievement goal that appears in the (suspended) intentions associated with events in C_E. For an agent ag and circumstance C, we have:

$$\text{INTEND}_{\langle ag,C\rangle}(\varphi) \equiv \varphi \in \bigcup_{i \in C_I} agls(i) \ \vee \ \varphi \in \bigcup_{\langle te,i\rangle \in C_E} agls(i).$$

Note that we are only interested in atomic formulæ at in triggering events that have the form of additions of achievement goals, and ignore all other types of triggering events. These are the formulæ that represent (symbolically) properties of the states of the world that the agent is trying to achieve (i.e., the intended states). However, taking such formulæ from the agent's set of intentions does not suffice for defining intentions, as there can be *suspended* intentions. Suspended intentions are precisely those that appear in the set of events.

We are now in a position to define the interpretation of the *desire* modality in AgentSpeak(L) agents. An agent in circumstance C desires a formula φ if, and only if, φ is an achievement goal in C's set of events C_E (associated with any intention i), or φ is a current intention of the agent; more formally:

$$\text{DES}_{\langle ag,C\rangle}(\varphi) \equiv \langle +!\varphi, i\rangle \in C_E \ \vee \ \text{INTEND}_{\langle ag,C\rangle}(\varphi).$$

Although this is not discussed in the original literature on AgentSpeak(L), it was argued in [3] that the *desire* modality in an AgentSpeak(L) agent is best represented by additions of achievement goals presently in the set of events, as well as its present intentions.

The definitions above tell us precisely how the BDI modalities that are used in claims about the system can be mapped onto the AgentSpeak(F) structures implemented as a Promela model. We next present, in full, the logical language that is used to specify properties of the BDI multi-agent systems written in AgentSpeak(F) that we can model-check following the approach in this paper.

The logical language we use here is a simplified version of $\mathcal{LORA}$ [17], which is based on modal logics of intentionality, dynamic logic, and CTL*. In the restricted version of the logic used here, we limit the underlying temporal logics to LTL rather than CTL*, given that LTL formulæ (excluding the "next" operator $\bigcirc$) are automatically translated into Promela never-claims by Spin. We describe later on some other restrictions aimed at making the logic directly translatable into Spin-format LTL formulæ.

Let pe be any valid Promela boolean expression, l be any agent label, x be a variable ranging over agent labels, and at and a be atomic and action formulæ defined in the AgentSpeak(F) syntax (see Section 3), except with no variables allowed. Then the set of well-formed formulæ (*wff*) of this logical language is defined inductively as follows:

1. pe is a *wff*;
2. at is a *wff*;
3. $(\mathsf{Bel}\ l\ at)$, $(\mathsf{Des}\ l\ at)$, and $(\mathsf{Int}\ l\ at)$ are *wff*;
4. $\forall x.(M\ x\ at)$ and $\exists x.(M\ x\ at)$ are *wff*, where $M \in \{\mathsf{Bel}, \mathsf{Des}, \mathsf{Int}\}$ and x ranges over a finite set of agent labels;
5. $(\mathsf{Does}\ l\ a)$ is a *wff*;
6. if φ and ψ are *wff*, so are $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \Rightarrow \psi)$, $(\varphi \Leftrightarrow \psi)$, always $(\Box\varphi)$, eventually $(\Diamond\varphi)$, until $(\varphi\ \mathcal{U}\ \psi)$, and "release", the dual of until $(\varphi\ \mathcal{R}\ \psi)$;
7. nothing else is a *wff*.

In the syntax above, agent labels denoted by l, and over which variable x ranges, are the ones associated with each AgentSpeak(F) program during the translation process. That is, the labels given as input to the translator form the finite set of agent labels over which the quantifiers are defined. The only unusual operator in this language is $(\mathsf{Does}\ l\ a)$, which holds if the agent denoted by l has requested action a and that is the next action to be executed by the environment. An AgentSpeak(F) atomic formula at is used to refer to what is actually true of the environment. In practical terms, it comes down to checking whether the predicate is in channel ***p*** where the percepts are stored by the (user-defined) environment. We do not give semantics (even informally) to the other operators above, as they have been extensively used in the multi-agent systems literature, and formal semantics can be found in the references given above. Note, however, that the BDI modalities can only be used with AgentSpeak(L) atomic propositions.

The concrete syntax used in the system for writing formulæ of the language above is that of Spin's LTL. Before passing the LTL formula on to Spin, we translate Bel, Des, and Int into expressions that access the AgentSpeak(L) data structures modelled in Promela (according to the definitions in the previous section). Modality Does is implemented by checking the first message in channel ***a***, the one used by agents to send to the environment the action they want to see executed (see Section 4.1). That first message in ***a*** is the action that is going to be executed next by the environment (as soon as it is scheduled by Spin).

5. A CASE STUDY

In this section, we describe a simplified auction scenario that we use to illustrate the sort of programs and specifications that can be used in our framework. The simple environment (written in Promela) announces 10 auctions and simply states which agent is the winner in each one (the one with the highest bid). There are three agents participating in these auctions, with three simplified bidding strategies. During the process of automatic translation from AgentSpeak(F) source files into a Promela code for the multi-agent system, we associate labels `ag1`, `ag2`, and `ag3` with the three agents present in the system. The AgentSpeak(F) source code for these agents is given below.

Agent ag1

```
+auction(N) : true
   <- place_bid(N,6).
```

Agent ag1 is a very simple agent which bids 6 whenever the environment announces a new auction.

Agent ag2

```
myself(ag2).
bid(ag2,4).
ally(ag3).

+auction(N) : myself(I) & ally(A) & not(alliance(A,I))
   <- ?bid(I,B); place_bid(N,B).

+auction(N) : alliance(A,I)
   <- place_bid(N,0).

+alliance(A,I) : myself(I) & ally(A)
   <- ?bid(I,B);
      .send(A,tell,bid(I,B));
      .send(A,tell,alliance(A,I)).
```

Agent ag2 bids 4, unless it has agreed on an alliance with ag3, in which case it bids 0. When ag2 receives a message from ag3 proposing an alliance, a belief alliance(ag3,ag2) is added to ag2's belief base (the default trust function is used). That is a triggering event to the last plan, which informs ag3 of how much ag2 was bidding, and confirms that ag2 agrees to form an alliance with ag3.

Agent ag3

```
myself(ag3).
bid(ag3,3).
ally(ag2).
threshold(3).

+auction(N) : threshold(T) & .gte(T,N)
   <- !bid_normally(N).

+auction(N) : myself(I) & winner(I)
             & ally(A) & not(alliance(I,A))
   <- !bid_normally(N).

+auction(N) : myself(I) & not(winner(I))
             & ally(A) & not(alliance(I,A))
   <- !alliance(I,A);
      !bid_normally(N).

+auction(N) : alliance(I,A)
   <- ?bid(I,B); ?bid(A,C);
      .plus(B,C,D); place_bid(N,D).

+!bid_normally(N) : true
   <- ?bid(I,B); place_bid(N,B).

+!alliance(I,A) : not(alliance(I,A))
   <- .send(A,tell,alliance(I,A)).
```

Agent ag3 tries to win the first T auctions, where T is a threshold stored in its belief base. If it is does not win any auctions up to that point, it will try to achieve an alliance with ag2 (by sending the appropriate message to it). When ag2 confirms that it agrees to form an alliance, then ag3 starts bidding, on behalf of them both, with the sum of their usual bids.

The following specifications were translated into Promela never-claims and used to verify certain (liveness) properties of the system.

$$\Box(\ \neg(\mathsf{Bel}\ \mathtt{ag3}\ \mathtt{winner}(\mathtt{ag3}))\wedge (\mathsf{Des}\ \mathtt{ag3}\ \mathtt{alliance}(\mathtt{ag3},\mathtt{ag2})) \Rightarrow \Diamond(\mathsf{Int}\ \mathtt{ag3}\ \mathtt{alliance}(\mathtt{ag3},\mathtt{ag2}))\ )$$

The specification above says that whenever ag3 does not believe it has won the auction and desires to form an alliance with ag2, then eventually ag3 will also intend to do so. This guarantees that AgentSpeak(L) will always find an applicable plan for handling the event +!alliance(ag3,ag2). The next specification assures that eventually both agents will have agreed on an alliance.

$$\Diamond(\ (\mathsf{Bel}\ \mathtt{ag2}\ \mathtt{alliance}(\mathtt{ag3},\mathtt{ag2}))\wedge (\mathsf{Bel}\ \mathtt{ag3}\ \mathtt{alliance}(\mathtt{ag3},\mathtt{ag2}))\ )$$

Further, we ensure that if they do agree on an alliance, then eventually ag3 will win all auctions on behalf of the alliance (it is true of the environment that ag3 is the winner).

$$\Box(\ (\mathsf{Bel}\ \mathtt{ag2}\ \mathtt{alliance}(\mathtt{ag3},\mathtt{ag2}))\wedge (\mathsf{Bel}\ \mathtt{ag3}\ \mathtt{alliance}(\mathtt{ag3},\mathtt{ag2})) \Rightarrow \Diamond\Box\mathtt{winner}(\mathtt{ag3})\ )$$

With these specifications, we detected an error in the AgentSpeak(F) code for ag3 as given above. The plan with triggering event +!alliance(I,A) is only applicable if an alliance has not yet been established. The context of the plan that generates that event guarantees that an alliance was not established at the time the event is generated. However, a run of the system is possible in which the message from ag2 confirming the alliance arrives after the generation of the event, but before the event is selected (in an attempt to find an applicable plan). Because in the current translation into Promela we cannot handle plan failure, this means that ag3 will not make a bid. Given that the environment was programmed so as not to proceed before all bids have been placed, the model checker finds an infinite cycle which does not satisfy the specifications. The easiest way to solve this problem, is by simply replacing not(alliance(I,A)) with true in the context of the plan for +!alliance(I,A). More than one message proposing an alliance may be sent to ag2, but the extra ones will be ignored by it, as it already believes alliance(ag3,ag2) (hence no further event is generated by belief revision). Once this modification is made, the formulæ above can be verified.

6. RELATED WORK

Since Rao's original proposal [13], a number of authors have investigated a range of different aspects of AgentSpeak(L). In [5], a complete abstract interpreter for AgentSpeak(L) was formally specified using the Z specification language. Some extensions to AgentSpeak(L) were proposed in [2], and an interpreter for the extended language was introduced. The extensions aim at providing a more practical programming language; the extended language also allows the specification of relations between plans and quantitative criteria for their execution. The interpreter then uses decision-theoretic task scheduling for automatically guiding the choices made by an agent's intention selection function.

In [11], an operational semantics for AgentSpeak(L) was given following Plotkin's structural approach; this is a more familiar notation than Z for giving semantics to programming languages. Later, that operational semantics was used in the specification of a framework for carrying out proofs of BDI properties of AgentSpeak(L) [3]. The particular combination of asymmetry thesis principles [15] satisfied by any AgentSpeak(L) agent was shown in that paper. This is relevant in assuring the rationality of agents programmed in AgentSpeak(L).

Model checking techniques have only recently begun to find a significant audience in the multi-agent systems community. Rao and Georgeff developed basic algorithms for model-checking BDI logics [14], but the authors proposed no method for generating BDI models from programs. In [1], a general approach for model-checking multi-agent systems was proposed, based on the branching temporal logic CTL together with modalities for BDI-like attitudes. However, once again no method was given for generating models from actual systems, and so the techniques given there could not easily be applied to verifying real multi-agent systems. In [8], techniques were given for model-checking temporal epistemic properties of multi-agent systems; the target of that work was the Spin model checker. However, that work did not consider an agent's motivational attitudes, such as desires and intentions.

Perhaps the closest work to ours is that in [18] on the MABLE multi-agent programming language and model-checking framework. MABLE is a regular imperative language (an impoverished version of C), extended with some features from Shoham's agent-oriented programming framework. Thus, agents in MABLE have data structures corresponding to beliefs, desires, and intentions, and can communicate using KQML-like performatives. MABLE is automatically translated into Promela, much like AgentSpeak(F) in this work. Claims about the system are also written in a $\mathcal{LORA}$-like language, which is also translated into Spin's LTL framework for model checking. The key difference is that MABLE is an imperative language, rather than a logic programming language inspired by PRS-like reactive planning systems, which is the case of AgentSpeak(F).

7. CONCLUSIONS

We have introduced a framework for the verification of agent programs written in an expressive logic programming language against BDI specifications. We do so by transforming AgentSpeak(F) code into Promela, and transforming BDI specifications into Spin-format LTL formulæ, then using Spin to model check the resulting system. AgentSpeak(L) is a practical BDI programming language with a well-defined theoretical foundation, and we here contribute to the missing aspect of practical AgentSpeak(L) verification.

The state of the art in model checking still requires the use of abstract versions of systems rather than direct implementations. As expected, during verification, our AgentSpeak(F) model in Promela is rather demanding on the Spin system in terms of memory and processing time. Future work should attempt at improving the efficiency of the AgentSpeak(F) model and devising suitable abstractoin techniques, so as to address scalability. Also, it would be interesting to add extra features to AgentSpeak(F) (e.g., handling plan failure, allowing first order terms, allowing variables in the specifications), as far as the complexity of model checking would allow.

We also plan as future work to verify more ambitious applications, such as autonomous spacecraft control (on the lines of [6]). Further, we are presently working on translating AgentSpeak(F) into Java rather than Promela, so that we can use JPF2 [16] rather then Spin for model checking. It would be interesting to compare the performances of Spin and JPF2 in model checking AgentSpeak(F) multi-agent systems.

Acknowledgements

This research was supported by a Marie Curie Fellowship of the European Community programme *Improving Human Potential* under contract number HPMF-CT-2001-00065. We are grateful for many suggestions for improvements made by Willem Visser and John Penix from NASA Ames Research Center during a visit to the University of Liverpool.

8. REFERENCES

[1] M. Benerecetti and A. Cimatti. Symbolic model checking for multi-agent systems. In *Proc. ECAI Workshop on Model Checking and Artificial Intelligence (MoChArt-2002), Lyon, France*, pages 1–8, 2002.

[2] R. H. Bordini, A. L. C. Bazzan, R. O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In C. Castelfranchi and W. L. Johnson (eds), *Proc. First International Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), Bologna, Italy*, pages 1294–1302. ACM Press, 2002.

[3] R. H. Bordini and Á. F. Moreira. Proving the asymmetry thesis principles for a BDI agent-oriented programming language. *Electronic Notes in Theoretical Computer Science*, 70(5), 2002.

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

[5] M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A formal computational model. *J. Logic and Computation*, 8(3):1–27, 1998.

[6] M. Fisher and W. Visser. Verification of autonomous spacecraft control — a logical vision of the future. In *Proc. Workshop on AI Planning and Scheduling For Autonomy in Space Applications, Manchester, UK*, 2002.

[7] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space craft controller using SPIN. *IEEE Trans. Software Engineering*, 27(8), Aug. 2001.

[8] W. Hoek and M. Wooldridge. Model checking knowledge and time. In D. Bošnački and S. Leue (eds), *Model Checking Software (LNCS Volume 2318)*, pages 95–111. Springer-Verlag: Berlin, Germany, 2002.

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

[10] R. Machado and R. H. Bordini. Running AgentSpeak(L) agents on SIM_AGENT. In J.-J. Meyer and M. Tambe, (eds), *Intelligent Agents VIII – Proc. Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), 2001, Seattle, WA*, number 2333 in LNAI, pages 158–174. Springer-Verlag, 2002.

[11] Á. F. Moreira and R. H. Bordini. An operational semantics for a BDI agent-oriented programming language. In *Proc. Workshop on Logics for Agent-Based Systems (LABS-02), held in conjunction with the Eighth International Conf. on Principles of Knowledge Representation and Reasoning (KR-2002), Toulouse, France*, pages 45–59, 2002.

[12] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote agents: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103:5–47, 1998.

[13] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram (eds), *Proc. Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands*, number 1038 in LNAI, pages 42–55. Springer-Verlag, 1996.

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

[15] A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *J. Logic and Computation*, 8(3):293–343, 1998.

[16] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. Fifteenth International Conf. on Automated Software Engineering (ASE'00), Grenoble, France*, pages 3–12. IEEE Computer Society, 2000.

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

[18] M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multi-agent systems with MABLE. In C. Castelfranchi and W. L. Johnson, (eds), *Proc. First International Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), Bologna, Italy*, pages 952–959. ACM Press, 2002.