Auton Agent Multi-Agent Sys (2006) 12: 239–256
DOI 10.1007/s10458-006-5955-7

Verifying multi-agent programs by model checking

Rafael H. Bordini · Michael Fisher · Willem Visser · Michael Wooldridge

Published online: 24 February 2006

Abstract This paper gives an overview of our recent work on an approach to verifying multi-agent programs. We automatically translate multi-agent systems programmed in the logic-based agent-oriented programming language AgentSpeak into either Promela or Java, and then use the associated Spin and JPF model checkers to verify the resulting systems. We also describe the simplified BDI logical language that is used to write the properties we want the systems to satisfy. The approach is illustrated by means of a simple case study.

Keywords Agent-oriented programming · AgentSpeak · Model checking · Spin · JPF

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, implementation, and verification of such systems. While such tools should be usable by a general computing audience, they should also have a strong theoretical underpinning, 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 [20,27].

R.H. Bordini (✉)
University of Durham, U.K
e-mail: R.Bordini@durham.ac.uk

M. Fisher · M. Wooldridge
University of Liverpool, U.K
e-mail: M.Fisher@csc.liv.ac.uk

M. Wooldridge
e-mail: M.J.Wooldridge@csc.liv.ac.uk

W. Visser
RIACS/NASA Ames Research Center, U.S.A
e-mail: wvisser@email.arc.nasa.gov

Currently, the most successful approach to the verification of computer systems against formally expressed requirements is that of *model checking* [12]. 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 [22,36].

Our aim in this paper is to present an overview of our recent work on the use of model checking techniques for the verification of systems implemented in AgentSpeak. Based on the work on reactive planning systems and the BDI agent architecture, Rao introduced in [30] an abstract agent-oriented programming language called originally AgentSpeak(L), which was later developed into a more practical programming framework [9]. While the theoretical foundations of AgentSpeak are increasingly well understood [10], there has been to date little research on the verification of AgentSpeak systems (or indeed any other agent-oriented programming language).

We begin by introducing AgentSpeak(F), a variant of AgentSpeak intended to permit its algorithmic verification. We then show how AgentSpeak(F) programs can be automatically transformed into Promela, the model specification language for the Spin model-checking system [23,22]. The translation from AgentSpeak(F) to Promela was first introduced in [4]. We also present an alternative approach, based on the translation of AgentSpeak agents into Java and verifying via JPF, a general purpose Java model checker [36]. Java models for the verification of AgentSpeak agents were first discussed in [7]. We also give here an example that illustrates our approach, and show results obtained using both alternative target model checkers.

In verification based on model checking, we need to provide a model of the system and also write down the properties that we require the system to satisfy. In the context of agent-oriented programming, ideally such properties would make use of modalities such as those of BDI logics [32,38]. In our approach, we use a simplified form of BDI logic in which specifications can be written; although it is much simpler than usual BDI logics, it is well suited for a (practical) agent programming language such as AgentSpeak. Also, the property specification language is defined so as to make it possible to transform specifications into Linear Temporal Logic (LTL) 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 by using existing sophisticated model checkers for LTL.

The set of tools derived from the work on both the Promela and Java alternatives is called CASP, and was briefly described in [11]. Elsewhere [6], we introduced a property-based slicing algorithm which can reduce the state space of the system's model that will be model checked without affecting the truth of the particular properties being verified. The whole approach was summarised in [5], which uses an interesting case study based on a NASA scenario (a autonomous Mars explorer).

Due to space limitations, we assume readers to be familiar with both AgentSpeak and Promela. The paper is structured as follows. Section 2 introduces the AgentSpeak(F) variant of AgentSpeak, and in the following sections we show how AgentSpeak(F) can be transformed into Promela models (Section 3), as well as into Java models (Section 4). Section 5 describes the BDI logical language used for specifications; that section also discusses how the BDI modalities are interpreted in terms of AgentSpeak data structures. We then present a case study in Section 6, discuss related work in Section 7, and finally we draw conclusions and mention future work.

2. AgentSpeak(F)

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

In order to ensure that systems to be model-checked are finite state, the maximum size of data structures and communication channels must be specified. In particular, Promela models require explicit bounds for all such structures. This means that, for a translator from AgentSpeak-like programs into a model checking system to work, a series of parameters stating the expected maximum number of occurrences of certain AgentSpeak 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 plan's context;
M_{Var}: maximum number of different *variables* used within 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 in the set of intentions; note that this is the number of plan instances rather than the number of intentions (each intention being a stack of plans);
M_{Act}: maximum number of *actions* requested by the agents that may have to wait to be executed by the environment;
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 currently given as input to the automatic translator, but they could be determined by purely syntactic pre-processing. The others are restrictions on the data structures used in an AgentSpeak interpreter, to be explained in Section 3. Some of these parameters will be used in the syntax of AgentSpeak(F), as seen below.

The grammar in Fig.1 gives the syntax of AgentSpeak(F). In that grammar, P stands for a predicate symbol and A for an action symbol. Terms t_i associated with them are either constants or variables rather than first order terms (cf. Prolog structures), as usual in AgentSpeak; the next section discusses this restriction further. As in Prolog, an uppercase initial letter is used for variables and lowercase for constants and predicate symbols (cf. Prolog atoms).

There are some special action symbols which are denoted by an initial '.' character (in extensions of the AgentSpeak language they have been referred to as *internal actions*). 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 inserted in the mailbox of agent l_2, having l_1 as sender, illocutionary force *ilf*, and propositional content

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

Fig. 1 The Syntax of AgentSpeak(F)

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 communication language [26]. 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 the appropriate (user-defined) *trust function* determines that the change is (socially) acceptable. There is one specific trust function for belief changes, and another for achievement goals. The latter defines informally some sort of subordination 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 is .print, which takes a string as parameter and is used to display messages; it has no effect on the internal structures of agents. Other pre-defined internal actions are, for example, used for conditional operators and arithmetic operations. Of course, in case users decide to make use of those operators, it is up to them to ensure finiteness of the models.

Syntactically, the main difference between AgentSpeak(F) and AgentSpeak(L) is that first order terms are not allowed, and there are given 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.

The current implementation of the set of tools generated as part of our work imposes some restrictions on certain features of AgentSpeak(L). In particular, it is presently not possible to use:

1. uninstantiated variables in triggering events;
2. uninstantiated variables in negated literals in a plan's context (as originally defined by Rao [30]);
3. the same predicate symbol with different arities (this only applies to the Promela version, not to the Java models);
4. 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, 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). Practical AgentSpeak interpreters allow 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.

3. Promela models of AgentSpeak(F) systems

We now describe how AgentSpeak(F) programs can be translated into Promela, the model specification language for the Spin model checker. Throughout this section, we presuppose some familiarity with Promela [22], as space restrictions prevent a detailed account here. For more detail on this translation, see [4].

A summary of the Promela model of an AgentSpeak(F) interpreter (i.e., for one agent) is shown in Fig. 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

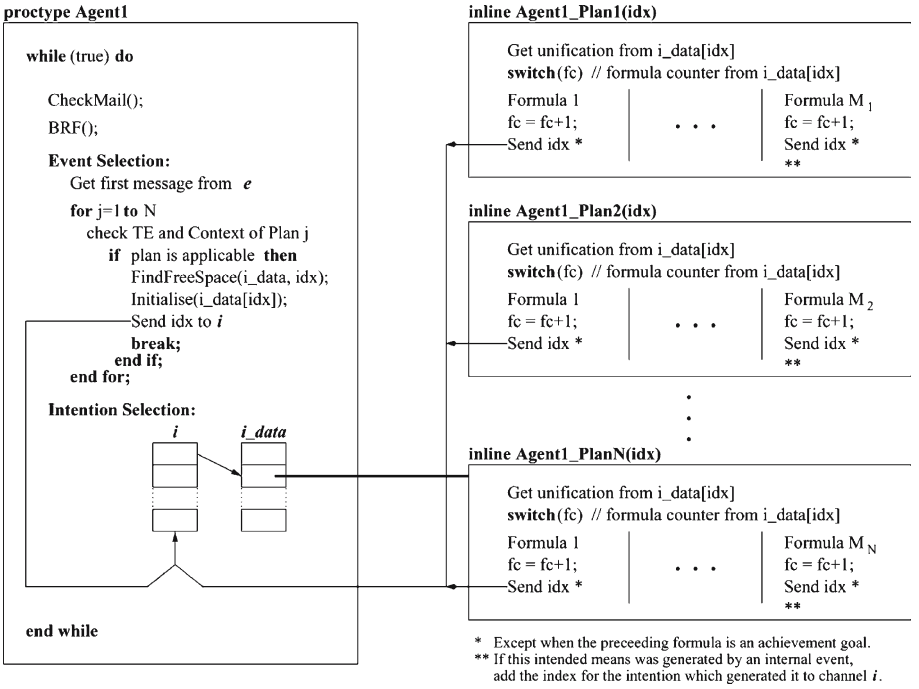

Fig. 2 Abstract Promela model for an AgentSpeak(F) agent

Promela does not support strings. AgentSpeak(F) variables are declared as integer Promela variables.[1]

3.1. 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 [20]. All such channels are described below.

Channel ***b*** represents the agent's *belief base*. The type for the messages 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 continue 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 execute the next formula of its body. 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 intention is represented here by the index to the entry in ***i_data*** that is associated with 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 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 there. 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, 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 detail on intention selection is given 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 entry in an array 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).

[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 identifiers.

[2] An AgentSpeak event is a tuple $\langle te, i \rangle$ where i is the intention that generated the triggering event te.

We can now discuss the implementation of each part of the reasoning cycle of an AgentSpeak(F) agent.

3.2. The interpretation cycle

The AgentSpeak(F) interpretation cycle is summarised in Fig. 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 2, 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 [16,30]. 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 2) 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 between agents is assumed.

The agent then runs its belief revision function ('BRF' in the figure). The function used, unless redefined by the user, 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 currently there. This generates corresponding belief-addition events (of format $\langle +at, \mathsf{T}\rangle$). 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., $\langle -at, \mathsf{T}\rangle$). It is, of course, a comparatively simple belief revision function, but quite appropriate for ordinary AgentSpeak 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.

Next, an event to be handled in this interpretation cycle has to be chosen. Events are currently handled via a FIFO policy. Thus, when new events are generated, they are appended to channel ***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 carries 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, which states 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' fc); 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 for that plan instance (this is stored in an array of size M_{Var}).

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

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

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. 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 scheduler is used unless an application-specific scheduler is given, it is straightforward to use a channel for this. 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 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 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 an agent's reasoning cycle, i.e., a cycle of interpretation of an AgentSpeak program. Each of the four main parts in the cycle (as seen in Fig. 2), namely belief revision , checking inter-agent communication messages, 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 `Select Event` and `SelectIntention` are initially defined as empty, 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 one in each of these channel) thus determining the event or intention that is going to be selected next.

The whole multi-agent system is initialised in the Promela `init` process. It runs the user-defined `Environment` process and waits for it to initialise channel ***p*** with the percepts that will be used in the first reasoning cycle of all agents. It then creates one process for each agent listed in the translation process.

4. Java models of AgentSpeak(F) systems

In addition to the AgentSpeak(F)-to-Promela translation approach, we have also developed an AgentSpeak(F)-to-Java translator; see [7] for more detail. The translation from AgentSpeak(F) to Java is fully automated; that is, it does not require any manual work from the users. Note, however, that this is for AgentSpeak(F) only, the restricted version of AgentSpeak presented in the Section 2.

Given the higher expressive power of Java, this translation is much simpler than the Promela translation. Translating to Java allows Java-based model checkers to be used to analyse the AgentSpeak(F) programs. We have used the JPF [24,36] Java model checker for analysis, but in future work we may also consider investigating the use of Bandera [14] and Bogor [33]. Furthermore, there is an efficient interpreter for an extended version of AgentSpeak called ***Jason*** [9] which allows the use of legacy code in Java within AgentSpeak programs. Using the translation to Java rather than Promela allows, in principle, systems that include code written in Java as well as AgentSpeak to be fully verified. Given the widespread use of Java in real-world applications, this approach is likely to be applied more generally than that based on Promela (which requires, e.g., Promela modelling skills, at least for defining a model of the environment).

Java PathFinder (JPF) [24,36] is an explicit state on-the-fly model checker that takes compiled Java programs (i.e., bytecode class-files) and analyses all paths through the program the bytecodes represent. The analysis includes checking for deadlocks, assertion violations, and linear time temporal logic (LTL) properties. An earlier version of JPF translated Java source code to Promela to allow model checking with Spin, whereas JPF analyses Java bytecodes directly with the aid of a special purpose Java Virtual Machine (JVM). JPF can therefore analyse any Java program, even one making heavy use of Java libraries—as is the case for the AgentSpeak(F) models used in the work described in this paper. However, because JPF works on the bytecode level, rather than an abstract model specification (as is the case with Promela), it is considerably slower than Spin.

In defining Java models of AgentSpeak(F) agents, we followed the main ideas used in generating a Promela model of the system. However, the Java model is far more elegant, as we can use instances of objects for various tasks such as creating instances of plans to be included in the set of intentions (an intention is a stack of partially instantiated plans). Also, in Java, we can handle such things as unification of logical terms in a much simpler way, and

even manage a clear 'plan library' which was not possible in Promela (where the resulting part of the model accounting for the plan library was very cumbersome).

Such flexibility in our Java models of AgentSpeak is possible partly because JPF works directly on Java bytecodes rather than, e.g., translating a subset of Java as in earlier versions of Bandera [14]. Therefore, in practice, any Java library (indeed anything in Java except native methods), including all the available classes for manipulating complex data structures (on which our AgentSpeak(F) models rely heavily), can be used in a systems to be model checked with JPF. As in ordinary Java programs, uses of such data structures do not require a static size given at compile time as for Promela. Of course, it is the responsibility of JPF users to ensure that the data structures never grow too much in size during execution otherwise the state space generated by the system will be too big for practical model checking. Still, not having to give specific bounds for the data structures in advance means that most of the translation parameters that we require in the translation to Promela are not necessary here, which also makes it significantly more practical for the users.

5. The property specification language

Ideally, we would like to be able to verify that systems implemented in AgentSpeak satisfy (or do not satisfy) properties expressed in a BDI-like logic. In this section, we show how BDI logic properties can be mapped down into Linear Temporal Logic (LTL) formulæ and associated predicates over the data structures in the Promela or Java models. This allows us to use existing LTL model checkers which have had continued development for many years, thus incorporating very sophisticated techniques to support the task of model checking, rather than developing a purpose-built model checker.

In [10], a way of interpreting the informational, motivational, and deliberative modalities of BDI logics for AgentSpeak agents was given, as part of a framework for proving properties of AgentSpeak based on its operational semantics. 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 plain LTL. The particular logical language that is used for specifying such properties is given towards the end of this section.

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 AgentSpeak(F)), and C is the agent's present circumstance defined as a tuple $\langle I, E, A, R, Ap, \iota, \rho, \varepsilon\rangle$. In a circumstance C, I is the set of intentions; E is the set of events that are yet to be handled by the agent; and A is a set of actions (it contains the actions that the agent decided to execute); the other components are not relevant here.

We here give only the main definitions from [10]; some of the motivations for the proposed interpretation is omitted. In particular, that paper discusses is more detail the interpretation of intentions and desires, as the *belief* modality is clearly defined in AgentSpeak. We say that an AgentSpeak agent ag, regardless of its circumstance C, believes a formula φ if, and only if, it is included in the agent's belief base; that is, for an agent $ag = \langle bs, ps\rangle$:

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

Note that a 'closed world' is assumed, so $\mathrm{BEL}_{\langle ag,C\rangle}(\varphi)$ is true if φ is included in the agent's belief base, and $\mathrm{BEL}_{\langle ag,C\rangle}(\neg\varphi)$ is true otherwise, where φ is an AgentSpeak atomic formula (i.e., *at* in Section 2). We next discuss the notion of intention, as it will be used in the definition of desire given later.

Before giving the formal definition for the *intention* modality, we first define an auxiliary function $agls : \mathcal{I} \rightarrow \mathbb{P}(\Phi)$, 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). Recall that a plan is syntactically defined as 'te `:` ct`<-`h`.`' (see Section 2). The *agls* function below returns all the achievement goals that appear within the triggering events of the plans in the intention given as argument:

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

Formally, we say an AgentSpeak 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. In the AgentSpeak interpreter, intentions may be suspended when they are waiting for an appropriate subplan to be chosen (in the form of an internal event, which is an event associated with an existing intention). Suspended intentions are, therefore, precisely those that appear in the set of events (for more detail on suspended intentions, see [16].

Now we can define the interpretation of the *desire* modality for AgentSpeak 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:

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

Although this is not discussed in the original literature on AgentSpeak(L), it was argued in [10] that the *desire* modality in an AgentSpeak agent is best represented by additions of achievement goals presently in the set of events, as well as its present intentions. Internal events in the form of achievement goals are clearly desires: traditionally, goals are defined as subset of desires which are mutually consistent [35]; in the simplified version of the architecture used for AgentSpeak, consistency of goals is taken for granted. Accordingly, the events in E that have the form of additions of achievement goals are desires; that is, the agent has not yet committed to a course of action to achieve that desire, but they represent states that the agent wishes to achieve. These desires may eventually become intentions as well, when the agent chooses a plan for handling that event; note that we here see intentions as a subset of an agent's desires.

The definitions above tell us precisely how the BDI modalities that are used in specifications of 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 $\mathcal{LORA}$ [39], which is based on modal logics of intentionality [13,32] dynamic logic, [19], and CTL* [1]. In the restricted version 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 some other restrictions aimed at making the logic directly translatable into 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 2), 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 atomic propositions.

The concrete syntax used in the system for writing formulæ of the language above is also dependent on the underlying model checker. Before we pass the LTL formula on to the model checker, we translate Bel, Des, and Int into code that checks the AgentSpeak data structures modelled in the model checker's input language (following to the definitions above). The Does modality is implemented by checking the first action in the environment's data structure where agents insert the actions they want to see executed by the process simulating the environment. That first item in such data structure is the action that is going to be executed next by the environment (as soon as it is scheduled for execution).

6. A case study

There have been three case studies illustrating the approach to verification of multi-agent systems described in this paper. In [4] we presented a case study based on a simplified auction

scenario. In [6], we used an interesting scenario based on typical situations in one day of activity of Mars autonomous rovers such as Sojourner (in the Mars Pathfinder mission) [3,37]. The development of autonomous rovers for planetary exploration is an important aim of the research on 'remote agents' carried out at space agencies [28].

In this paper, in order to illustrate our approach and to give an example of model checking statistics (time and memory used) generated when using Spin and JPF as target model checkers, we shall summarise the scenario and results reported in [17]. The scenario used is that of cleaning robots, which appears frequently in the Agents literature. Please refer to that paper for detailed explanations of the AgentSpeak code and BDI properties used in the examples below.

There are two robots in this scenario. Robot `r1` searches for pieces of garbage and when one is found, the robot picks it up, takes it to the location of `r2`, drops the garbage there, and returns to the location where it found the garbage and continues its search from that position. Robot `r2` is situated at an incinerator; whenever garbage is taken to its location by `r1`, `r2` just puts it in the incinerator. One or two pieces of garbage are randomly scattered on the grid. Another source of non-determinism is a certain imprecision of the robot's arm that grabs the pieces of garbage. The action of picking up garbage may fail, but it is assumed that the mechanism is good enough so that it never fails more than twice; that is, in the worst case robot `r1` has to attempt to pick up a piece of garbage three times, but by then `r1` will definitely have grabbed it. The territory to be cleared of pieces of garbage is abstractly represented here as a finite 2D grid. The AgentSpeak(F) code for `r1` is given in Fig. 3.

Agent `r2` is defined by a very simple AgentSpeak(F) program. All it does is to burn the pieces of garbage (action `burn(garb)`) when it senses that there is garbage on its location. A belief `+garbage(r2)` is added to the agent's belief base by belief revision (from the perception of the environment) when that is case. The only plan in `r2`'s program is therefore `+garbage(r2) : true <- burn(garb).`

The environment simulating the territory to be cleaned was written both as a Promela process and a Java class. A matrix of size 5×5 is used for that purpose. Two pieces of garbage were scattered randomly in the territory (using non-determinism in the model). Whenever the environment changes (due to an agent performing an action), the percepts are updated with the agents' positions and corresponding facts about the presence of garbage in the current position of each agent.

When programming an agent's practical reasoning, it is difficult to ensure that such an agent will behave as expected in all circumstances, hence the importance of verification in the context of agent-oriented programming. In the first implemented version of this scenario, we did not skip `r2`'s position when the basic action `next(slot)` was requested by `r1`. This created a series of inconsistent behaviours when the robot was passing by `r2`'s position while searching for garbage to be collected. As usual, ordinary programming errors can be detected too. For example, in an earlier version of the AgentSpeak(F) code, plan p2 had an empty context, which made the agent again act inconsistently when passing by a slot with garbage in it, while on its way to deliver a piece of garbage to `r1` and back (i.e., when there was garbage in its trajectory between another dirty slot and `r2`). Of course, such configuration of the Mars territory was considered by the model-checker, which showed the problem clearly. In our experience, it also happens that specifications that seem to be correct, turn out not to be, shedding light on the details of the application scenario. An example of such situation is given below. Below, we list six specifications that were used in the verification of the system used in this case study.

Agent r1

Beliefs

```
pos(r2,2,2).
checking(slots).
```

Plans

```
+pos(r1,X1,Y1) : checking(slots) & not(garbage(r1))          (p1)
   <- next(slot).
+garbage(r1) : checking(slots)                               (p2)
   <- !stop(check);
      !take(garb,r2);
      !continue(check).
+!stop(check) : true                                         (p3)
   <- ?pos(r1,X1,Y1);
      +pos(back,X1,Y1);
      -checking(slots).
+!take(S,L) : true                                           (p4)
   <- !ensure_pick(S);
      !go(L);
      drop(S).
+!ensure_pick(S) : garbage(r1)                               (p5)
   <- pick(garb);
      !ensure_pick(S).
+!ensure_pick(S) : true <- true.                             (p6)
+!continue(check) : true                                     (p7)
   <- !go(back);
      -pos(back,X1,Y1);
      +checking(slots);
      next(slot).
+!go(L) : pos(L,Xl,Yl) & pos(r1,Xl,Yl)                       (p8)
   <- true.
+!go(L) : true                                               (p9)
   <- ?pos(L,Xl,Yl);
      moveTowards(Xl,Yl);
      !go(L) :
```

Fig. 3 AgentSpeak(F) code for robot r1

$$\Box((\mathsf{Bel}\ \texttt{r1}\ \texttt{garbage(r1)}) \Rightarrow \Diamond \texttt{garbage(r2)}) \tag{1}$$

$$\Box((\texttt{pos(r1,2,2)} \wedge (\mathsf{Does}\ \texttt{r1}\ \texttt{drop(garb)})) \Rightarrow \Diamond(\mathsf{Des}\ \texttt{r1}\ \texttt{go(back)})) \tag{2}$$

$$\Diamond((\mathsf{Int}\ \texttt{r1}\ \texttt{take(garb,r2)}) \wedge \Diamond(\mathsf{Does}\ \texttt{r1}\ \texttt{drop(garb)})) \tag{3}$$

$$\Diamond((\mathsf{Int}\ \texttt{r1}\ \texttt{continue(check)}) \wedge (\mathsf{Bel}\ \texttt{r1}\ \texttt{checking(slots)})) \tag{4}$$

$$\Box((\mathsf{Des}\ \texttt{r1}\ \texttt{continue(check)}) \Rightarrow \Diamond(\mathsf{Does}\ \texttt{r1}\ \texttt{next(slot)})) \tag{5}$$

$$\Box((\mathsf{Does}\ \texttt{r1}\ \texttt{next(slot)}) \Rightarrow (\neg\texttt{pos(r1,2,2)} \wedge (\mathsf{Bel}\ \texttt{r1}\ \texttt{checking(slots)}))) \tag{6}$$

Specification (1) says that it is always the case that, if r1 perceives garbage on its location, then eventually it will be true of the environment that there is garbage in r2's location. Although specification (1) appeared to us as a valid specification of the system, unforeseen

situations could happen in which that property would not hold. It is possible that `r2` incinerates the garbage in its position while `r1` is still there. Before `r1` proceeds with belief revision, it still believes there is garbage in its position (the same position as `r2`, where `r1` itself took the piece of garbage it encountered). However, if that was the last piece of garbage in the whole territory, at that point it is true that `r1` believes `garbage(r1)`, yet it is not true that eventually there will be garbage in `r2`'s position (as that one still believed by `r1` no longer exists). Apart from property (1), all other specifications were fully verified by both model-checkers and are thus valid properties of the system.

Some statistics produced by Spin and JPF when verifying Specification (4), using the current (preliminary) version of the Promela and Java models, are as follows. In Spin, the state space had 333,413 states, verification used 210.51 MB of memory, and took nearly 65.78 seconds to complete. In JPF, there were 236,946 states, verification used 366.68 MB of memory, and took 18:49:16 hours to complete. However, in another setting of the scenario, where garbage is placed at fixed positions (1, 1) and (3, 3), the verification took JPF 76.63 seconds to finish, and 5.25 seconds for Spin. Although from this results it appears that JPF is not scaling as well as Spin, it must be said that these experiments were run using an older version of JPF than the latest one now available *open source* [24]. Besides, we are currently in the process of making some of the Java classes used AgentSpeak models native to the JPF model checker; effectively, this corresponds to developing a specialised version of JPF for model checking AgentSpeak systems in particular. Preliminary experiments indicate that such specialised version of JPF will increase the efficiency of model checking our agent programs by at least an order of magnitude.

7. Related work

Since Rao's original proposal [30], a number of authors have investigated a range of different aspects of AgentSpeak. In [16], a complete abstract interpreter for AgentSpeak was formally specified using the Z specification language. Most of the elements in that formalisation had already appeared in [15]; this highlights the fact that AgentSpeak is strongly based on the experience with the BDI-inspired dMARS system [25]. Various extensions of AgentSpeak have been appeared in the literature, and an interpreter for the extended language is available [9].

In [10], an operational semantics for AgentSpeak was given following Plotkin's [29] structural approach; this is a more familiar notation than Z for giving semantics to programming languages. The operational semantics was used in the specification of a framework for carrying out proofs of BDI properties of AgentSpeak; that work was used in Section 5 to show precisely how the BDI modalities used in specifications to be verified against models of AgentSpeak-like agents are interpreted here.

Model checking techniques have only recently begun to find a significant audience in the multi-agent systems community. Rao and Georgeff [31] developed basic algorithms for model-checking BDI logics, but the authors proposed no method for generating BDI models from programs. In [2], 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 [21], 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 [39] 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 [34]. 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).

8. 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 either Promela or Java, and transforming BDI specifications into LTL formulæ, then using either Spin or JPF to model check the resulting system. AgentSpeak is a practical BDI programming language with a well-defined theoretical foundation, and our work contributes to the (as yet missing) aspect of practical verification of AgentSpeak multi-agent system, an aspect that is increasingly important as multi-agent systems start to be used for the development of dependable applications.

In future work, we plan to improve the efficiency of the models by optimisations on the Promela and Java code that is automatically generated. We may also consider the implementation of a custom-made model checker for AgentSpeak(F), or more particularly in customising JPF for the use of models generated from AgentSpeak (as mentioned earlier, by making some of the Java classes used in AgentSpeak(F) models native to JPF). Further, it would be interesting to add extra features to the languages we use for agent verification (e.g., handling plan failure and allowing first order terms), although we need to consider the effects that these will have in the state space of the generated models.

Another line of work we plan continue in the future is on further state-space reduction techniques for AgentSpeak, a line of work we started in [6]. That paper introduced a property-based slicing algorithm for AgentSpeak. Property-basedslicing is a precise form of under approximation, which means that we can do model checking more efficiently yet knowing that the reduced state-space will not affect the properties being verified; the experiments reported in that paper included doing model checking for the original system and the sliced version of the system, showing a significant reduction of the state space. With the use of state-space reduction techniques, much larger applications can be verified. We also plan as future work to verify real-world applications, particularly in the area of autonomous spacecraft control, on the lines of [18].

It is still far from clear whether we will be able to address satisfactorily issues such as openness of multi-agent systems (i.e., when any number of heterogeneous agents may interact at a given time) and evolving agents (i.e., coping with emergent phenomena). Although it is early days, our view is that combining deductive [17] and algorithmic techniques (as those presented here), as well as continued work on state-space reduction techniques suitable for multi-agent systems in particular, will have an important role in the verification of such types of multi-agent systems in the future.

Acknowledgements This research was supported by Marie Curie Fellowships of the EC programme *Improving Human Potential* under contract number HPMF-CT-2001-00065.

References

1. Allen Emerson, E. (1990). Temporal and modal logic. In J. van Leeuwen, (Ed.) *Handbook of theoretical computer science*, (vol. B, Chap. 16, pp. 997–1072). Amsterdam: Elsevier Science.
2. Benerecetti, M., & Cimatti, A. Symbolic model checking for multi-agent systems. In *Proceedings of the model checking and artificial intelligence workshop (MoChArt-2002), held with 15th ECAI, 21–26 July, Lyon, France*, (pp. 1–8).
3. Biesiadecki, J., Maimone, M. W., & Morrison, J. (2001). The Athena SDM rover: A testbed for marsrover mobility. In *Sixth international symposium on AI, robotics and automation in space (ISAIRAS-01), June, Montreal, Canada*.
4. Bordini, R. H., Fisher, M., Pardavila, C., & Wooldridge, M. (2003). Model checking AgentSpeak. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, & M. Yokoo (Eds.), *Proceedings of the second international joint conference on autonomous agents and multi-agent systems (AAMAS-2003), Melbourne, Australia, 14–18 July*, (pp. 409–416) New York, NY, ACM Press.
5. Bordini, R. H., Fisher, M., Visser, W. & Wooldridge, M. (2004a). Model checking rational agents. *IEEE Intelligent Systems, 19*(5), 46–52.
6. Bordini, R. H., Fisher, M., Visser, W., & Wooldridge, M. (2004b) State-space reduction techniques in agent verification. In N. R. Jennings, C. Sierra, L. Sonenberg & M. Tambe (Eds.). *Proceedings of the third international joint conference on autonomous agents and multi-agent systems (AAMAS-2004), New York, NY, 19–23 July* (pp. 896–903) New York, NY, ACM Press.
7. Bordini, R. H., Fisher, M., Visser W., & Wooldridge, M (2004c) Verifiable multi-agent programs. In M. Dastani, J. Dix, & A. El Fallah-Seghrouchni, (Eds.) *Programming multi-agent systems, proceedings of the first international workshop (ProMAS-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia (Selected Revised and Invited Papers)*, number 3067 in Lecture notes in artificial intelligence, (pp. 72–89) Berlin, Springer-Verlag.
8. Bordini, R. H., Hübner, J. F., et al. (2005) ***Jason***. `http://jason.sourceforge.net/`.
9. Bordini, R. H., Hübner, J. F., & Vieira, R. (2005). ***Jason*** and the golden fleece of agent-oriented programming. In R. H. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni, (Eds.) *Multi-agent programming: Languages, Platforms and applications*, chap. 1 Springer-Verlag.
10. Bordini, R. H., & Moreira, Á. F. (2004 September). Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence, 42* (1–3), 197–226. Special issue on computational logic in multi-agent systems.
11. Bordini R. H., Visser W., Fisher, M., Pardavila, C., & Wooldridge, M. (2003). Model checking multi-agent programs with CASP. In W.A. Hunt Jr. & F. Somenzi, (Eds.) *Proceedgins of the fifteenth conference on computer-aided verification (CAV-2003), Boulder, CO, 8–12 July*, number 2725 in Lecture Notes in Computer Science, (pp. 110–113) Berlin: Springer-Verlag. Tool description.
12. Clarke, E. M., Grumberg, O., & Peled, D. A. (2000). *Model checking*. Cambridge, MA: The MIT Press.
13. Cohen, P. R., & Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42(3), 213–261.
14. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, S. L., & Zheng, H. (2000, June) Bandera : Extracting finite-state models from java source code. In *Proceedings of the 22nd international conference on software engineering*, Limeric, Ireland: ACM Press.
15. d'Inverno, M., Kinny, D., Luck, M., & Wooldridge, M. (1998). A formal specification of dMARS. In M. P. Singh, A. S. Rao, & M. Wooldridge, (Eds.), *Intelligent agents IV—proceedings of the fourth international workshop on agent theories, architectures, and languages (ATAL-97), providence, RI, 24–26 July, 1997*, number 1365 in Lecture notes in artificial intelligence, (pp. 155–176). Berlin: Springer-Verlag.
16. d'Inverno, M. & Luck, M. (1998). Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation, 8*(3), 1–27.
17. Fisher, M. (2005, January) Temporal development methods for agent-Based systems. *Journal of Autonomous Agents and Multi-Agent Systems, 10*(1), 41–66.
18. Fisher, M., & Visser, W. (2002). Verification of autonomous spacecraft control—a logical vision of the future. In *Proceedings of the workshop on AI planning and scheduling for autonomy in space applications, co-located with TIME-2002, 7–9 July, Manchester, UK*.
19. Harel, D., Kozen, D., & Tiuryn, J. (2000). *Dynamic logic*. Cambridge, MA: The MIT Press.
20. Havelund, K., Lowry, M., & Penix, J. (2001, August). Formal analysis of a space craft controller using SPIN. *IEEE Transactions on Software Engineering, 27*(8).

21. van der Hoek, W., & Wooldridge, M. (2002). Model checking knowledge and time. In D. Bošnački, & S. Leue, (Eds.) *Model checking software, proceedings of SPIN 2002 (LNCS Volume 2318)*, (pp. 95–111). Berlin, Germany: Springer-Verlag.
22. Holzmann, G. (1997, May). The Spin model checker. *IEEE Transaction on Software Engineering, 23*(5), 279–295.
23. Holzmann, G. J. (1991). *Design and validation of computer protocols*. Prentice Hall.
24. Java PathFinder. `http://javapathfinder.sourceforge.net`.
25. Kinny, D. (1993). The distributed multi-agent reasoning system architecture and language specification. Technical report, Melbourne, Australia: Australian Artificial Intelligence Institute.
26. Labrou, Y., & Finin, T. (1994, November). A semantics approach for KQML—a general purpose communication language for software agents. In *Proceedings of the third international conference on information and knowledge management (CIKM'94)*. ACM Press.
27. Muscettola, N., Nayak, P. P., Pell, B., & Williams B. C. (1998a) Remote agents: To boldly go where no AI system has gone before. *Artificial Intelligence, 103*, 5–47.
28. Muscettola, N., Nayak, P. P., Pell, B., & Williams, B. C. (1998b, August) Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence, 103*(1–2), 5–47.
29. Plotkin, G. D. (1981). A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, Aarhus.
30. Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde & J. Perram, (Eds.) *Proceedings of the seventh workshop on modelling autonomous agents in a multi-agent world (MAAMAW'96) 22–25 January, Eindhoven, The Netherlands*, number 1038 in Lecture notes in artificial intelligence (pp. 42–55) London. Springer-Verlag.
31. Rao, A. S., & Georgeff, M. P. (1993) A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the thirteenth international joint conference on artificial intelligence (IJCAI-93)* (pp. 318–324) Chambéry, France.
32. Rao, A. S., & Georgeff, M. P. (1998). Decision procedures for BDI logics. *Journal of Logic and Computation, 8*(3), 293–343.
33. Robby, M. Dwyer, B., & Hatcliff, J. (2003, March). Bogor: An extensible and highly-modular model checking framework. In *In the proceedings of the fourth joint meeting of the european software engineering conference and ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE)*.
34. Shoham, Y. (1990). Agent-oriented programming. Technical report STAN–CS–1335–90, computer science department, Stanford University, Stanford, CA 94305.
35. Singh, M. P., Rao, A. S., & Georgeff, M. P. (1999) Formal methods in DAI: Logic-based representation and reasoning. In G. Weiß, (ed.) *Multiagent systems—a modern approach to distributed artificial intelligence*, Chap. 8 (pp. 331–376) Cambridge, MA: MIT Press.
36. Visser, W., Havelund, K., Brat, G., & Park, S. (2000). Model checking programs. In *Proceedings of the fifteenth international conference on automated software engineering (ASE'00), 11–15 September, Grenoble, France* (pp. 3–12) IEEE Computer Society.
37. Washington, R., Golden, K., Bresina, J., Smith, D. E., Anderson, C., & Smith, T. (1999). Autonomous rovers for mars exploration. In *Aerospace conference, 6–13 March, Aspen, CO*, Vol. 1 (pp. 237–251) IEEE.
38. Wooldridge, M. (2000) *Reasoning about rational agents*. Cambridge, MA: The MIT Press.
39. Wooldridge, M., Fisher, M., Huget, M.-P., & Parsons, S. (2002). Model checking multiagent systems with MABLE. In C. Castelfranchi & W. L. Johnson (Eds.) *Proceedings of the first international joint conference on autonomous agents and multi-agent systems (AAMAS-2002), 15–19 July, Bologna, Italy* (pp. 952–959) New York, NY: ACM Press.