Verifiable Multi-Agent Programs

Rafael H. Bordini* Michael Fisher* Willem Visser† Michael Wooldridge*

* University of Liverpool, Liverpool L69 7ZF, U.K.
{R.Bordini,M.Fisher,M.J.Wooldridge}@csc.liv.ac.uk

† RIACS/NASA Ames Research Center
Moffett Field, CA 94035, U.S.A.
wvisser@email.arc.nasa.gov

ABSTRACT

AgentSpeak is a reactive planning language for programming autonomous agents. It has recently been shown that model checking techniques can be applied to the verification of AgentSpeak systems, through a translation to PROMELA, the model specification language for the SPIN LTL model-checking system. In this paper, we introduce an alternative verification approach for AgentSpeak, by translating AgentSpeak to Java and then applying JPF2, a general purpose Java model checker. The primary advantage of this approach is that Java is the language of choice for most agent implementations, and the approach is thus much closer to the current practice of agent development than the PROMELA-based approach. We examine both alternatives by means of a practical application, provide a qualitative comparison between them, and identify some key issues for future research.

Keywords

Multi-Agent Systems, Agent-Oriented Programming, Model Checking

1. INTRODUCTION

In order to provide generic development tools for rational agents, a number of rational agent programming languages are now being developed, for example by extending conventional programming languages with capabilities from the BDI (Belief-Desire-Intention) theory of agency [20, 23]. Such languages provide high-level abstractions that aid the construction of dynamic, autonomous components, together with the deliberation that goes on within them. AgentSpeak(L) is an influential example of such a language [18]. AgentSpeak(L) is essentially a logic programming language with abstractions provided for key aspects of rational agency, such as beliefs, goals and plans.

As agent programming agents are increasingly deployed, the *verification* of agent-based systems — showing that a system is correct with respect to its specification — is a correspondingly important issue, especially as agent systems are applied to safety-critical applications such as autonomous spacecraft control [16]. However, there is so far a lack of verification tools specifically designed with *rational* agents in mind, and this issue is the main focus of this paper. Currently, the most successful approach to the verification of computer systems against formally expressed requirements is that of *model checking* [5]. Model checking was originally developed for verifying that finite state concurrent systems implement specifications expressed in temporal logic. Historically, model-checking techniques have been most widely applied to the verification of hardware systems, but recently they have been increasingly used in the verification of software systems and communications protocols [11, 22].

Our aim in this paper is to consider model checking techniques for verifying systems implemented in AgentSpeak(L). In [3], preliminary model-checking techniques for AgentSpeak(L) were reported. That work was based on a translation from AgentSpeak(L) to PROMELA, the finite state model specification language for the SPIN LTL model checker [11]. In this paper, we present an alternative approach, based on the translation of AgentSpeak(L) agents into Java and verifying via a general purpose Java model checker [22]. We present an example scenario, with comparative qualitative results obtained for the two approaches. Thus, this work forms the basis for increasingly sophisticated rational agent verification techniques to be developed in the future, particularly through translation to Java.

This paper is structured as follows. The next section presents AgentSpeak(L), a BDI logic programming language inspired by reactive planning systems. Section 3 presents our approach to programming and verifying multi-agent systems. It presents the AgentSpeak(F) language, a finite state version of AgentSpeak(L), and it presents a simplified BDI logic in which properties to be verified against AgentSpeak(F) multi-agent systems can be written. Our approach allows us to take advantage of existing LTL model-checkers; in Section 4, we briefly recall our previous work on using SPIN [11], but we also mention a new alternative, which is to use JPF2 [22]. We describe a practical experience with model checking an AgentSpeak(F) system in Section 5; we use here a scenario with two garbage collecting robots.

That section also briefly compares SPIN and JPF2. Finally, we discuss related work in Section 6 and discuss future work and conclusions in Section 7.

2. AgentSpeak(L)

The AgentSpeak(L) programming language was introduced in [18]. It is a natural extension of logic programming for the BDI agent architecture, and provides an elegant abstract framework for programming BDI agents. In this paper, we only give a brief introduction to AgentSpeak(L); see [18, 4] 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 "$e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots ; h_n.$" 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). An agent's definition also include three selection functions: $\mathcal{S}_\mathcal{E}$ selects an event from the set of events; $\mathcal{S}_\mathcal{O}$ selects an option or an applicable plan from a set of applicable plans; and $\mathcal{S}_\mathcal{I}$ selects an intention from the set of intentions (the chosen intention is then executed).

3. VERIFYING AgentSpeak(F) SYSTEMS

Recall that our main goal in this research is to facilitate model checking of AgentSpeak(L) 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(L) to finite state systems: the result is AgentSpeak(F), a finite state version of AgentSpeak(L). This language was first defined in [3] and is briefly presented here so that we can later give an example of our approach to programming and verifying multi-agent systems. The idea is to translate multi-agent systems defined in this language into the input language of existing model checkers, so that we can take advantage of the extensive range of tools for model checking that are available.

Further, we would like to be able to verify that systems implemented in AgentSpeak(L) satisfy (or do not satisfy) properties expressed in BDI logic [20]. These logics formalise all the main concepts of the BDI architecture used in reactive planning systems such as those generated by AgentSpeak(L) agents. This section also presents a simplified form of BDI logic which we are able to convert into Linear Temporal Logic (LTL) formulæ, so that we can use existing LTL model-checkers for verifying our multi-agent systems.

The purpose of this section is to introduce the features and limitations of the languages used in our approach to code the system and produce the specifications it should satisfy. Their use will be made clear in Section 5, where we present a case study.

3.1 AgentSpeak(F)

The grammar in Figure 1 gives the concrete syntax of AgentSpeak(F): 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 (cf., Prolog atoms). Note that first order terms (cf., Prolog structures) are not allowed in the present version of AgentSpeak(F); we discuss this restriction further later on.

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 $\mathsf{.send}(l_2, ilf, at)$, a message will be inserted in the mailbox of agent l_2, having l_1 as sender, illocutionary force ilf, and propositional con-

```
ag  ::=  bs  ps
bs  ::=  at1 .   ...   atn .
at  ::=  P(t1, ..., tn)
ps  ::=  p1   ...   pn
p   ::=  te : ct <- h .
te  ::=  +at   |   -at
      |  +g    |   -g
ct  ::=  at    |   true
      |  not ( at )
      |  ct1 & ... & ctn
h   ::=  A(t1, ..., tn)
      |  g   |   u   |   h ; h
g   ::=  !at   |   ?at
u   ::=  +at   |   -at
```

Figure 1: The Concrete Syntax of AgentSpeak(F).

tent *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 [14]. 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. The restrictions of AgentSpeak(F) in relation to AgentSpeak(L) are as follows:

- uninstantiated variables in triggering events;
- uninstantiated variables in negated literals in a plan's context (as originally defined by Rao [18]);
- 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);
- first order terms (rather than just constants and variables).

The following restrictions of AgentSpeak(F) in relation to AgentSpeak(L) are necessary (in particular for the translation into PROMELA): (i) no uninstantiated variables are allowed in triggering events; (ii) no uninstantiated variables are allowed in negated literals in a plan's context (as originally defined by Rao [18]); (iii) the same predicate symbol cannot be used with different arities; (iv) first order terms are not allowed (terms are only 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 [13], 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). On the second restriction, we point out that 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), so the second restriction is not an unreasonable one.

Finally, we remark that the multi-agent system is specified by the user as a collection of AgentSpeak(F) source files, one for each agent in the system. The user can change various predefined functions which are part of the interpretation of AgentSpeak(L) agents. Also, the user has to provide the environment where the agents will be situated; this must be done in the model language of the model checker itself rather than AgentSpeak(F).

3.2 Specifying BDI Properties

In the context of verifying multi-agent systems implemented in AgentSpeak(L), the most appropriate way of specifying the properties that the system satisfy (or do not satisfy) is expressing those properties using BDI logics [20]. In this section, we show how simple BDI logical properties can be mapped down into Linear Temporal Logic (LTL) formulæ and associated predicates over the AgentSpeak(L) data structures in the system.

In [4], a way of interpreting the informational, motivational, and deliberative modalities of BDI logics for AgentSpeak(L) agents was given; this is based on the operational semantics of AgentSpeak(L). This framework was used to prove which of the asymmetry thesis principles [20] are enforced by AgentSpeak(L). This is relevant in assuring the rationality of agents programmed in AgentSpeak(L). In this work, we use that same framework for interpreting the BDI modalities in terms of data structures within the model of an AgentSpeak(F) agent given in the model checker input language. This way, we can to translate (temporal) BDI properties into LTL formulæ. The particular logical language that is used for specifying such properties is given next.

The logical language we use here is a simplified version of $\mathcal{LORA}$ [23], 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 $\bigcirc$) can be automatically processed by our target model-checkers. Other restrictions, aimed at making the logic directly translatable into LTL formulæ, are described below.

Let *pe* be any valid boolean expression in the model specification language of the model checker being used, *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 3.1), 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, this amounts to checking whether the predicate is in the data structure where the percepts are stored by the 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.

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 predicates accessing the AgentSpeak(F) data structures modelled in the model checker's input language (according to the definitions in the previous section). 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 environment process 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).

4. USING EXISTING MODEL CHECKERS

Previous Work on Using SPIN

In [3], we showed how to convert programs in AgentSpeak(F) into PROMELA, so that SPIN could be used in model-checking AgentSpeak(F) systems. In fact, the restrictions imposed on AgentSpeak(F) in relation to AgentSpeak(L) were aimed at facilitating the generation of PROMELA models of BDI agents. In order to ensure that AgentSpeak(F) multi-agent systems to be model checked could be converted into PROMELA, 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, it requires a series of parameters stating the expected maximum number of entries in certain data structures used in an AgentSpeak(L) interpreter. The list of all parameters needed by our automatic translator is given in [3]. Having a PROMELA model of the AgentSpeak(F) multi-agent system and a LTL formula representing the original BDI specification (following the approach mentioned in Section 3.2), we can then use SPIN to formally verify our systems.

JPF2

Java PathFinder 2 (JPF2) [22] 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 for deadlock, assertion violations and linear time temporal logic (LTL) properties. An earlier version of JPF2, called JPF1, translated Java source code to PROMELA to allow model checking with SPIN, whereas JPF2analyses Java bytecode directly with the aid of a special purpose Java Virtual Machine (JVM). JPF2 can therefore analyse any Java program, even one making heavy use of Java libraries — as is the case for the AgentSpeak(F) model used here. However, because JPF2 works on the bytecode level, rather than an abstract model specification (as is the case with PROMELA), it is considerably slower than SPIN.

Java Models of AgentSpeak(F) Agents

In order to use JPF2 as the underlying model-checker, we need to translate our AgentSpeak(F) multi-agent systems into Java code rather than PROMELA. Obviously, this is a much easier task, as PROMELA is a very limited language (as it is intended for the verification of simple communication protocols). While we followed the main ideas used in generating a PROMELA model of the system [3], the Java model is *much* more elegant, as we can use instances of objects (such as plans) in representing the set of intentions (a set of partially instantiated plans). Also, in Java, we can handle such things as unification of logical terms is a much simpler way, and even manage a proper "plan library" which was not possible in PROMELA (where the resulting part of the model accounting for the plan library was very cumbersome).

Our AgentSpeak(F) model in PROMELA was already known to be very demanding in terms of memory and processing time needed during verification by SPIN, and similar problems occurred with JPF2. In Section 7 we mention how we plan to tackle this problem in future work. Meanwhile, in order to reduce the state space generated by the model checkers, we have fixed an order in which the agents run, and each is given one AgentSpeak(L) reasoning cycle in turn (agents' execution is, of course, interspersed with the execution of the environment). This is not ideal as it slightly changes the semantics of the AgentSpeak(L) system, but note that some existing agent-based systems actually simulate multi-agent systems in this very same way (it simply means giving a strictly fair distribution of computational resources, as one agent cannot perform more reasoning cycles than the others in a given period of time). This change was made both in the PROMELA and in the Java models of AgentSpeak(F).

5. MODEL CHECKING IN PRACTICE

To the best of our knowledge, this is the first study where two model checkers are compared in the verification of the same high-level agent description. In this section, we report a practical exercise in model-checking multi-agent systems following the approach described in the previous section where both JPF2 and SPIN have been used. The scenario used here is clearly a toy problem, but it shows an important aspect of multi-agent systems, which is the natural abstraction that results from separating high-level reasoning from the details of acting in an environment. The idea is to verify that the general reasoning of agents are correct, before moving to implementation.

5.1 The Mars Scenario

The scenario used here involves two robots that are collecting garbage on the planet Mars. Robot `r1` searches for pieces of garbage and when one is found, the robot picks it up, take it to the location of `r2`, drops the garbage there, and return 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 AgentSpeak(F) code for `r1` is given in Figure 2. In the figure, we have annotated each plan with a label, so that we can refer to them in the text below.

The only initial beliefs this agent needs is on the position of agent `r2` on the "grid" into which the territory is divided, and that, to start with, what it is doing is to check all the slots in the territory grid for garbage. All the plans are explained below.

Plan `p1` is used when the agent perceives that it is in a new position and its currently checking for garbage. If there is no garbage perceived in that slot, all it has to do is a basic action `next(slot)` which moves the robot to the next slot in the grid, except the one where the incinerator is: garbage in that position is dealt with by robot `r2`. Note that this is a basic action from the point of view of the agent's reasoning: it is assumed that the robot has the mechanisms to move itself to a next position on the territory grid, automatically skipping the incinerator's position.

The environment provides percepts stating whether a piece of garbage is present in `r1` or `r2`'s position. When `r1` perceives garbage in its position, a belief `garbage(r1)` is added to its belief base, hence plan `p2` can be used (if the agent is still in its operation mode where it is checking for garbage, rather then moving them to `r2` or coming back). The task of dealing with a perceived piece of garbage is accomplished in three parts. It involves achieving subgoals which: (i) make sure that the robot will stop checking for garbage in a consistent manner (e.g., remembering where it was so that the task of continuing the search can be resumed); (ii) actually taking the garbage (`garb`) to `r2`; and (iii) resuming the task of checking the slots for garbage. Each of these goals are achieved by the following three plans respectively.

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).
```

Figure 2: AgentSpeak(F) Code for Robot `r1`.

When the agent intends to achieve subgoal (i) above, plan `p3` is its only option, and it is always applicable (i.e., it has an empty context). The agent retrieves from its belief base its own current position (which will be in its belief base from perception of the environment). It then makes a note to itself of where is the position it will need to go back to when it is to resume searching for garbage. This is done by adding a belief to its belief base: `+pos(back,X1,Y1)`. It also removes from its belief base the information that it is presently searching for garbage, as this is no longer true (it will soon intend to take the garbage to `r2` and then go back).

Subgoal (ii) is achieved by plan `p4`, which states that in order for the robot to take garbage to `r2`, it should pick it up, then achieve the subgoal of going to the position where `r2` is, and when `r1` is there, it can finally drop the garbage. Note that `pick(garb)` and `drop(garb)` are again basic actions, i.e., things that are assumed the robot can "physically" perform in the environment, by means of its hardware apparatus.

Plans `p5` and `p6` together ensure that the robot will keep trying to pick up a piece of garbage until it can no longer perceive garbage on the grid (i.e., the grabbing mechanism succeeded). Recall that the grabbing mechanism is imprecise, so the robot may need to try a few times before it succeeds.

Plan p7 is used for the agent to continue the task of checking the grid slots for garbage. The agent needs to achieve the subgoal of going back to its previous position (`!go(back)`), and once there, it can remove the note it has made of that position, remember itself that it is now again checking the slots for garbage, and then proceed to the next slot.

The last two plans are used for achieving the goal of going to a specific position on the grid where L is located. Plan p9 retrieves the belief the agent has about the location of reference L, then move itself towards those coordinates `moveTowards(X1,Y1)`, and keep going towards L (this is a recursive goal). Plan p8 provides the end of the recursion, saying that there is nothing else to do in order to achieve the goal of going towards L, if agent r1 is already at that position.

Agent r2 is defined by the very simple AgentSpeak(F) code below. All it does is to burn the pieces of garbage when it senses that there is garbage on its location.

Agent r2
```
+garbage(r2) : true
   <- burn(garb).
```

All it does is to burn the pieces of garbage (`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).

The environment simulating the Mars 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.

We now turn to the practical experience with the scenario. We comment on the corrections we had to make to arrive to the code above, and the specifications we know it now does (or does not) satisfy.

5.2 Experimental Results

Verification is particularly important for agent-oriented languages. Because we are specifying an agent's high-level reasoning only, it is difficult to ensure that such agent will behave as expected in all circumstances. 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 removed. 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 on it, when on its way to deliver a piece of garbage to r1 and back (i.e., when there was garbage in one of 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. The following specifications were used in the verification of our system:

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

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

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

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

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

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

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 believe 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 Mars 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). All other specification were fully verified by both model-checkers and are thus valid properties of the system.

Some statistics produced by SPIN and JPF2 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 JPF2, 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 JPF2 76.63 seconds to finish, and 5.25 seconds for SPIN. It appears that JPF2 is not presently scaling as well as SPIN.

Note that we do not aim, in this paper, to compare in detail the performance of both model-checkers; figures are given just as a reference for the scale of time and space required in this application. However, some of the advantages of each model-checker are discussed next.

5.3 Comparing SPIN and JPF2

At present, SPIN can handle larger state spaces than JPF2, and is also faster, as seen by the results above. However, the input language of JPF2 (standard Java code) is far more interesting for our purposes than PROMELA. First, the PROMELA model of an AgentSpeak(F) agent is very large and cumbersome (because PROMELA is too impoverished a language for the sophisticated interpretation needed by AgentSpeak(L)-like languages). Second, Java is used as a basis of many agent platforms, and it is the language that has been used the most in actual implementation of multi-agent system. It is thus more likely that this latter alternative will be used in practice by developers of multi-agent systems. Finally, abstraction techniques used with JPF2 may provide a way to model-check AgentSpeak(L) programs directly, thus avoiding finiteness restrictions.

Therefore, at present, there are trade-offs involved in choosing which target notation/model checker to use: one can translate to a notation which is cumbersome but with a powerful model checker attached, or one can translate to a more suitable target notation, but with a slower model checker. This may point out towards the need to write a custom-made model checker: in this case it would not require complex translations and could be much faster. In summary, if on one hand SPIN was much faster in model-checking our Mars robot scenario, on the other hand the PROMELA translator took much longer to implement than the Java one, it also has many restrictions that can be easily dealt with in Java (e.g., the bounds on the size of the date structures), and the generated PROMELA model is so cumbersome that it is barely readable.

6. RELATED WORK

Since Rao's original proposal [18], a number of authors have investigated a range of different aspects of AgentSpeak(L). In [7], an abstract interpreter for AgentSpeak(L) was formally specified using the Z specification language. Most of the elements in that formalisation had already appeared in [6]; this highlights the fact that AgentSpeak(L) is strongly based on the experience with the BDI-inspired dMARS system [12]. 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 [15], an operational semantics for AgentSpeak(L) was given following Plotkin's structural approach [17]; 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) [4]. This was used in this work to define precisely how the BDI specifications to be verified against systems of multiple AgentSpeak(L)-like agents are interpreted. In [4], The particular combination of asymmetry thesis principles [20] satisfied by any AgentSpeak(L) agent was shown in that paper. This is relevant in assuring the rationality of agents programmed in AgentSpeak(L).

In [10], 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 [24] 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 [21]. 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. shorter version of the above: MABLE is automatically translated into PROMELA and claims are written in a $\mathcal{LORA}$-like language that is translated into LTL, similarly to the work presented in [3]. The key difference is that MABLE is an imperative language, rather than a logic programming language inspired by PRS-like reactive planning systems, as in the approach presented here.

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 either PROMELA (as reported in [3]) or Java, and transforming BDI specifications into LTL formulæ, then using either SPIN or JPF2 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.

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 [19], 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.

The statistics of our case study in Section 5.2 show that our AgentSpeak(F) model in PROMELA and Java are highly demanding in terms of memory and processing time. Future work will attempt 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), and we plan to combine our approach with deductive verification so that we can handle larger applications. Further, it would be interesting to add extra features to our approach to agent verification (e.g., handling plan failure, allowing first order terms, allowing variables in the property specifications), as far as the complexity of model checking would allow. Finally, we also plan as future work to verify more ambitious applications, such as autonomous spacecraft control (along the lines of [9, 8]).

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.

8. REFERENCES

[1] M. Benerecetti and A. Cimatti. 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*, 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, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), 15–19 July, Bologna, Italy*, pages 1294–1302, New York, NY, 2002. ACM Press.

[3] R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia, 14–18 July*, 2003. To appear.

[4] R. H. Bordini and Á. F. Moreira. Proving the asymmetry thesis principles for a BDI agent-oriented programming language. In J. Dix, J. A. Leite, and K. Satoh, editors, *Proceedings of the Third International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-02), 1st August, Copenhagen, Denmark*, Electronic Notes in Theoretical Computer Science 70(5). Elsevier, 2002. <http://www.elsevier.nl/locate/entcs/volume70.html>. CLIMA-02 was held as part of FLoC-02.

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

[6] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. S. Rao, and M. Wooldridge, editors, *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 LNAI, pages 155–176. Springer-Verlag, Berlin, 1998.

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

[8] M. Fisher and C. Ghidini. The ABC of rational agent modelling. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), 15–19 July, Bologna, Italy*, pages 849–856, New York, NY, 2002. ACM Press.

[9] M. Fisher and W. Visser. 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*, 2002.

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

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

[12] D. Kinny. The distributed multi-agent reasoning system architecture and language specification. Technical report, Australian Artificial Intelligence Institute, Melbourne, Australia, 1993.

[13] R. Machado and R. H. Bordini. Running AgentSpeak(L) agents on SIM_AGENT. In J.-J. Meyer and M. Tambe, editors, *Intelligent Agents VIII – Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), August 1–3, 2001, Seattle, WA*, number 2333 in LNAI, pages 158–174, Berlin, 2002. Springer-Verlag.

[14] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II—Proceedings of the Second International Workshop on Agent Theories, Architectures, and Languages (ATAL'95), held as part of IJCAI'95, Montréal, Canada, August 1995*, number 1037 in LNAI, pages 347–360, Berlin, 1996. Springer-Verlag.

[15] Á. F. Moreira and R. H. Bordini. An operational semantics for a BDI agent-oriented programming language. In *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS-02), held in conjunction with the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), April 22–25, Toulouse, France*, pages 45–59, 2002.

[16] 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.

[17] G. D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, Aarhus, 1981.

[18] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *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 LNAI, pages 42–55, London, 1996. Springer-Verlag.

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

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

[21] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[22] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the Fifteenth International Conference on Automated Software Engineering (ASE'00), 11-15 September, Grenoble, France*, pages 3–12. IEEE Computer Society, 2000.

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

[24] M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multi-agent systems with MABLE. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), 15–19 July, Bologna, Italy*, pages 952–959, New York, NY, 2002. ACM Press.