Model Checking Multi-Agent Programs with CASP

Rafael H. Bordini[1], Michael Fisher[1], Carmen Pardavila[1], Willem Visser[2], and Michael Wooldridge[1]

[1] Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, U.K.
{R.Bordini, M.Fisher, C. Pardavila, M.J.Wooldridge}@csc.liv.ac.uk
[2] RIACS/NASA Ames Research Center, Moffett Field, CA 94035, U.S.A.
wvisser@email.arc.nasa.gov

1 Introduction

In order to provide generic development tools for rational agents, a number of agent programming languages are now being developed, often by extending conventional programming languages with capabilities from the BDI (Belief-Desire-Intention) theory of rational agency [7, 9]. Such languages provide high-level abstractions that aid the construction of dynamic, autonomous components, together with the deliberation that goes on within them. One particularly influential example of such a language is AgentSpeak(L) [6], a logic programming language with abstractions provided for key aspects of rational agency, such as beliefs, goals and plans.

Model checking techniques have only recently begun to find a significant audience in the multi-agent systems community. In particular, our approach is the first whereby model checking can be applied to a logic programming language aimed at reactive planning systems following the BDI architecture.

Our aim in this paper is to describe a toolkit we have developed to support the use of model checking techniques for AgentSpeak(L). The toolkit, called CASP (Checking AgentSpeak Programs), automatically translates AgentSpeak(L) code into the input language of existing model checkers. In [1], we showed how to translate from AgentSpeak(L) to Promela, the model specification language for the SPIN LTL model checker [5]. More recently [2], we developed an alternative approach, based on the translation of AgentSpeak(L) agents into Java and verification via JPF2, a general purpose Java model checker [8].

2 AgentSpeak(L)

The AgentSpeak(L) programming language was introduced in [6]. It is a natural extension of logic programming for the BDI agent architecture, and provides an elegant abstract framework for programming BDI agents. The BDI architecture is, in turn, the predominant approach to implementing "intelligent" or "rational" agents [9]. 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 termed *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 and test goals are predicates (as for beliefs) prefixed with operators '`!`' and '`?`' respectively. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, these initiate the execution of *subplans*.) A *test goal* returns a unification for the associated predicate with one of the agent's beliefs; they fail otherwise. A *triggering event* defines which events may initiate the execution of a plan. An *event* can be internal, when a subgoal needs to be achieved, or external, when generated from belief updates as a result of perceiving the environment. 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 used to distinguish them. If e is a triggering event, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n$ are goals or actions, then "e `:` b_1 `&` ... `&` b_m `<-` h_1`;` ...`;` h_n`.`" is a *plan*. A plan is formed by a *triggering event* (denoting the purpose for that plan), followed by a conjunction of belief literals representing a *context* (they are separated by '`:`'). The context must be a logical consequence of that agent's current beliefs for the plan to be *applicable*. The remainder of the plan (after '`<-`') is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan, if applicable, is chosen for execution.

```
+concert(A,V) : likes(A)
   <- !book_tickets(A,V).

+!book_tickets(A,V)
   :  not(busy(phone))
   <- call(V); ...;
      !choose_seats(A,V).
```

Fig. 1. Examples of Plans

Figure 1 shows some example AgentSpeak(L) plans. They tell us that, when a concert is announced for artist `A` at venue `V` (so that, from perception of the environment, a belief `concert(A,V)` is *added*), then if this agent in fact likes artist `A`, then it will have the new goal of booking tickets for that concert. The second plan tells us that whenever this agent adopts the goal of booking tickets for `A`'s performance at `V`, if it is the case that the telephone is not busy, then it can execute a plan consisting of performing the basic action `call(V)` (assuming that making a phone call is an atomic action that the agent can perform) followed by a certain protocol for booking tickets (indicated by '...'), which in this case ends with the execution of a plan for choosing the seats for such performance at that particular venue.

3 Property Specification Language

In the context of verifying multi-agent systems implemented in AgentSpeak(L), the most appropriate way of specifying the properties that the system satisfies (or does not satisfy) is expressing those properties within BDI logics [7, 9]. In our framework, we can express simple BDI logical properties that can be subsequently translated into Linear Temporal Logic (LTL) formulæ (as used by SPIN and JPF2) with associated predicates over AgentSpeak(L) data structures.

Our property specification language includes the standard BDI modal operators Bel (Belief), Des (Desire), and Int (Intention); however, these can only be applied to AgentSpeak(L) atomic formulæ. The language also includes a modality used to refer to an agent performing an action in the environment (called Does). For example, for an agent i as in Figure 1, one can express properties such as $\Box((\mathsf{Des}\ i\ \texttt{book\_tickets(a1,v1)}) \Rightarrow (\mathsf{Bel}\ i\ \texttt{likes(a1)}))$ and $\Box((\mathsf{Int}\ i\ \texttt{book\_tickets(a1,v1)}) \Rightarrow \Diamond(\mathsf{Does}\ i\ \texttt{call(v1)}))$. Note that an intention requires an applicable plan; in BDI theory, intentions are desired states of affair which an agents has committed itself to achieve (in practice, through the execution of a plan). Further details of the language can be found in [1].

4 Practical Model Checking

In [1], we defined a finite-state version of AgentSpeak(L), called AgentSpeak(F), and we showed how to convert a set of AgentSpeak(F) programs into PROMELA, as well as how to convert BDI properties into LTL formulæ (following the translation approach mentioned above). We can then use the SPIN model checker to verify multi-agent systems written in AgentSpeak(F). Recently, we introduced an alternative approach where AgentSpeak(F) programs are translated to Java code, thus allowing the use of JPF2 for model checking [2]. Note that before model checking can start, one also needs to encode the environment where the agents are to be situated in the input notation of the model checker being used.

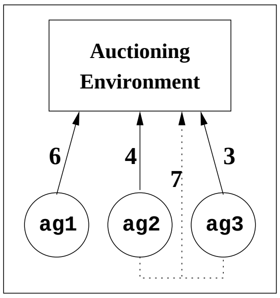

Fig. 2. Auction System

One of the case studies we carried out to assess our approach was the analysis of a simplified auction scenario, illustrated in Figure 2. A simple environment announces 10 auctions and states which agent is the winner in each one (the one with the highest bid). There are three agents, written in AgentSpeak(F), participating in these auctions. Agent `ag1` is a very simple agent which bids 6 whenever the environment announces a new auction. Agent `ag2` bids 4, unless it has agreed on an alliance with `ag3`, in which case it bids 0. Agent `ag3` tries to win the first `T` auctions, where `T` is a threshold stored in its belief base. If it does not win any auctions up to that point, it will try to achieve an alliance with `ag2` (by sending the appropriate message to it). When `ag2` confirms that it agrees to form an alliance, then `ag3` starts bidding, on behalf of them both, the sum of their usual bids (i.e., 7).

Initial results have indicated that, while Java provides a much more appropriate target language than PROMELA, JPF2 does not scale as well as SPIN. Java is the language of choice in most practical implementations of mulit-agent systems, and the Java model is much more clear and easily extensible; JPF2 also handles unbounded data structures, so we do not have to limit them during translation time. We have used both model checkers for verifying that the system described above satisfies the following specifications (among others):

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

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

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

5 Ongoing and Future Work

We are currently attempting to improve the efficiency of the AgentSpeak(F) models by optimisations on the PROMELA or Java code that is automatically generated. We are also working on a deeper analysis of the advantages and disadvantages of those model checkers in the verification of AgentSpeak(F) systems.

As future work, we intend to examine symbolic model checking for AgentSpeak(F), possibly by using NuSMV2 [3]. We also plan to combine our present 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 specifications). Finally, we also plan as future work to verify more ambitious applications, such as autonomous spacecraft control (along the lines of [4]).

Acknowledgements: This work has been partially supported by a Marie Curie fellowship of the EC, contract HPMF-CT-2001-00065 ("Model Checking for Mobility").

References

1. R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. *2nd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems*, 2003.
2. R. H. Bordini, W. Visser, M. Fisher, and M. Wooldridge. Model checking a reactive planning language for multi-agent systems. Submitted, 2003.
3. A. Cimatti *et al.*. NuSMV2: an opensource tool for symbolic model checking. *14th Int. Conf. on Computer Aided Verification*, LNCS 2404, Springer-Verlag, 2002.
4. M. Fisher and W. Visser. Verification of autonomous spacecraft control — a logical vision of the future. *Workshop on AI Planning and Scheduling For Autonomy in Space Applications, co-located with TIME*, 2002.
5. G. J. Holzmann. The Spin model checker. *IEEE Transaction on Software Engineering*, 23(5):279–295, May 1997.
6. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. *7th MAAMAW Workshop*, LNAI 1038, London, 1996. Springer-Verlag.
7. A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–343, 1998.
8. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *15th Int. Conf. on Automated Software Engineering*. IEEE Computer Society, 2000.
9. M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA, 2000.