A Flexible Framework for Verifying Agent Programs*

(Short Paper)

Louise A. Dennis
Dept. of Computer Science
University of Liverpool, UK
l.a.dennis@liverpool.ac.uk

Berndt Farwer
Dept. of Computer Science
Durham University, UK
{berndt.farwer,r.bordini}@durham.ac.uk

Rafael H. Bordini
Dept. of Computer Science
Durham University, UK
{berndt.farwer,r.bordini}@durham.ac.uk

Michael Fisher
Dept. of Computer Science
University of Liverpool, UK
mfisher@liverpool.ac.uk

ABSTRACT

There is an increasing number of agent-oriented programming languages that have working interpreters and platforms, with significant progress in the quality of such platforms over the last few years. With these platforms becoming more popular, and multi-agent systems being increasingly used for safety-critical applications, the need for verification techniques that apply to systems written in such languages is proportionally intensified. Building on our previous work on model checking for a particular agent-oriented programming language, we have developed a new approach whereby model checking techniques can be used directly on a variety of such languages. The approach also supports the verification of multi-agent systems where individual agents have been programmed in different agent languages.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Languages, Verification

Keywords

Agent-Oriented Programming Languages, BDI Agents, Verification, Model Checking, Java Pathfinder

1. INTRODUCTION

The last decade has seen significant growth in both the amount and maturity of research being carried out in the area of *agent-based systems*. An agent can be seen as an *autonomous* computational entity, making its own decisions about what activities to pursue. *Rational agents* make such decisions in a rational and explainable way and, since agents are autonomous, understanding *why* an agent chooses a course of action is vital. Therefore, a key new aspect in the design and analysis of such systems is the need to consider not just what agents do but *why* they do it.

So, as agent-based solutions are used in increasingly complex and critical areas, there is greater need to analyse, comprehensively, the behaviour of such systems. Not surprisingly, therefore, formal verification techniques tailored specifically for agent-based systems is an area that is now attracting a great deal of attention.While program verification is well advanced, for example Java verification using Java PathFinder [10, 15], verification of agent-oriented programs poses new challenges that have not yet been adequately addressed, particularly in the context of practical model-checking tools. In agent verification, we have to verify not only what the agent does, but *why* it chose that course of action, *what* the agent believed that made it choose to act in this way, and what its intentions were in doing so.

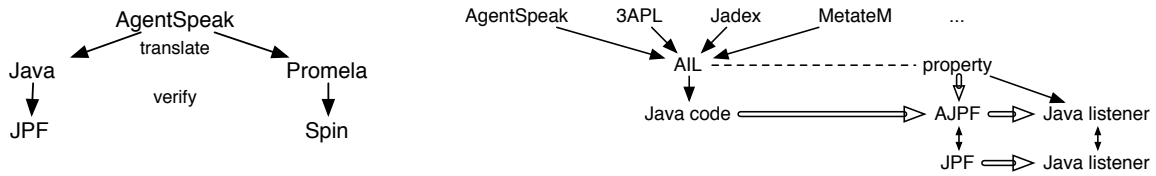
Rather than providing an approach for *one* particular programming language, we here describe a novel architecture for a system allowing the verification of a wide range of agent-based programs, produced using various high-level agent programming languages. Although the implementation of this architecture is a complex undertaking, we here provide the first comprehensive account of the architecture.

Our previous work [1, 3] has concentrated on model checking techniques for agent-based systems written in the logic-based agent-programming language AgentSpeak [13]. As described above, it is vital to verify not only the behaviour that the agent systems has, but to verify *why* the agents are undertaking certain courses of action. Thus, the temporal basis of model-checking captures the *dynamic* nature of agent computation, but is extended with *intensional modal operators* capturing the *informational* ('beliefs'), *motivational* ('desires') and *deliberative* ('intentions') aspects of a rational agent. Such pioneering work on *model checking* techniques for the verification of agent-based systems has appeared, for example, in [1, 11, 3, 12].

The structure of the paper is as follows. In Section 2, we give an overview of our Agent Infrastructure Layer (AIL) followed, in Section 3, by a description of how we are incorporating existing BDI languages into our framework. We motivate the key characteristics of our property specification language and discuss our extension AJPF (Agent Java PathFinder) of Java PathFinder (JPF) in Section 4. In the concluding section, we summarise our contribution and indicate future research directions.

*Work supported by EPSRC grants EP/D054788 (Durham) and EP/D052548 (Liverpool).

Cite as: A Flexible Framework for Verifying Agent Programs (Short Paper), Louise A. Dennis, Berndt Farwer, Rafael H. Bordini and Michael Fisher, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons(eds.),May,12-16.,2008,Estoril,Portugal,pp. 1303-1306.
Copyright © 2008, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.



(a) Old approach

(b) New approach

Figure 1: Approaches for Model Checking Agent Programming Languages.

2. ARCHITECTURE OF THE AIL

Building on our previous work, we have developed a new framework that brings a large part of the verification-related aspects together. Figure 1(a) shows diagrammatically how the previous approach worked. Contrast this with the new approach outlined in this paper (see Figure 1(b)).

One of the problems with existing approaches to model checking agent programs is that one had to find ways of encoding beliefs, goals, etc., within the state of the JPF or Spin state machine. This is a complex task, and one that would need to be (at least partly) done again to allow model checking of different programming languages. As in other fields of Computer Science, with the emergence of various modelling formalisms (in particular here new agent programming languages), a need for a unifying framework arises.

We have investigated the key aspects underlying several BDI programming languages [5]. Based on that, we have developed the Agent Infrastructure Layer (AIL), a collection of Java classes that: (i) enables implementation of AIL interpreters for various agent languages, (ii) contains adaptable, clear semantics, and (iii) can be verified through AJPF, an extended version of the open source Java model checker JPF [15]. AJPF is a customisation of JPF that was optimised for AIL-based interpreters.

The AIL can be viewed as a platform on which agents programmed in different programming languages co-exist, and together with AJPF this provides uniform model checking techniques for various agent-oriented programming languages. This is further extended with the *MCAPL* [1] *interface* which allows programming languages that do not have their own AIL-based interpreters to be model checked against specifications written in the same property specification language. (However, these will not benefit from the efficiency improvements that the optimised AJPF provides.)

The AIL is not intended as a new language in its own right, but as an intermediate layer incorporating the main features of practical agent languages. We identify the key *operations* that many (BDI-)languages use and treat these operations as part of an *AIL toolkit*. The semantic rules in [5] are a part of this toolkit but any given language can use only some of these rules in its own AIL-based interpreter and may choose to add its own custom rules built from the basic operations made available. These operations and rules have formal semantics and are implemented in Java.

We assume that agents in an agent programming language all possess a *reasoning cycle* consisting of several (≥ 1) stages (a reasoning cycle can often be broken down into various identifiable stages that help formalisation and understanding). Each stage is a disjunction of rules that define how an agent's state may change during the execution of that stage. The combined rules of the stages of the reasoning cycle define the operational semantics of that language. The construction of an interpreter for a language involves the implementation of these rules (which in some cases might simply make reference to the pre-implemented rules) and a reasoning cycle. This means that the AIL can be viewed as a collection of Java classes/methods that provide the building blocks for custom programming of agent language interpreters, with the particular advantage of making model checking possible (and more efficient). In this way, we can implement, for example, an AgentSpeak interpreter following the AgentSpeak operational semantics but using the AIL operations rather than using Java from scratch.

Common to all language interpreters implemented using AIL methods are the AIL-agent data structures for beliefs, intentions, goals, etc., which are accessed by the model checker and on which the modalities of the property specification language are defined. The implicit data structures of a given BDI language need to be translated into the AIL's data structures. In particular the initial state of an agent has to be translated into an AIL agent state.

In addition to the AIL toolkit, we also provide a set of interface classes, which we call the *MCAPL interface*. As mentioned earlier, this allows agents to be model checked using the same property specification language even if no AIL-based interpreter for that language has been developed. An implementation of the MCAPL interface must define the required operators of the property specification language. For instance, agents implementing the MCAPL agent interface must provide a method which succeeds when they believe the given parameter (represented as a "formula") is true. In this way, the implementation of such a method effectively implements the semantics for *belief* in that specific supported language. The AIL implements these interfaces and so defines an AIL-specific semantics for the property specification language; supported languages that use the AIL must ensure that their use of the AIL makes the semantics of the properties consistent with their own semantics of those modalities.

Figure 2 provides a diagrammatical representation of AIL. An agent, originally programmed in some agent programming language 'APL' and running on the AIL platform, is represented in the figure. It uses AIL data structures to store its internal state comprising, for instance, a belief base, a plan library, a current intention, and a set of intentions (as well as other temporary state information). It also uses the specific interpreter for the programming language 'APL' that is built using AIL classes and methods. The interpreter defines the reasoning cycle for 'APL' which interacts with the model checker, essentially notifying it when a new state is reached that is relevant for verification.

[1] *Model Checking Agent Programming Languages.*

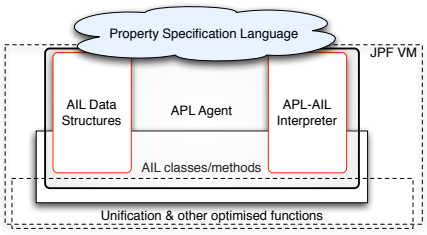


Figure 2: Overview of the AIL Architecture.

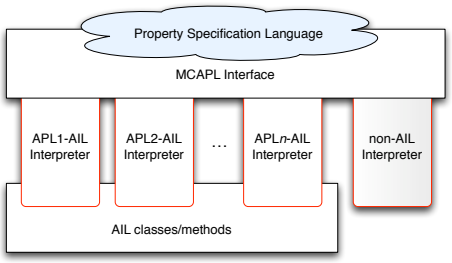


Figure 3: The MCAPL Interface.

The agent runs in the JPF virtual machine. This is a Java virtual machine specially designed to maintain backtrack points and explore, for instance, all possible thread scheduling options (that can affect the result of the verification) [15]. The JPF model checker is extensible and configurable, which will allow us to optimise its performance for AIL-based systems.

3. INCORPORATING BDI LANGUAGES

Translating programs for use in the AIL framework requires the actual language to be adapted to the AIL classes and a translation to be carried out on the initial state of the agents as described by the program. To adapt a language, we use the AIL operations and rules as building blocks for custom programming language interpreters. That way, we can implement, for example, a 3APL interpreter following the 3APL operational semantics but using AIL operations. We do not have an AIL reasoning cycle as such and each language has its customised interpreter reimplementing the original reasoning cycle of the respective language.

The AIL provides operations and rules to ease the implementation of language interpreters. These building blocks can be used to recreate the operational semantic rules of the target language. Using these Java classes and methods provided in AIL makes programming the language interpreter rather straightforward. The AIL provides all the infrastructure that is needed for a full implementation of an agent programming language so that the only task required to implement a language on top of AIL is the plugging together of AIL operations to form the specific semantic rules. Taking into consideration that the operations include all the basic querying and updating of agent state components, the rules are easily constructed.

Some special methods of the AIL are provided in the expectation that they will be overridden by the custom language interpreter. This allows the pre-existing operational semantic rules provided by AIL to be customised to a large extent by simply overriding key methods without requiring the creation of new language-specific rules. The translation of the initial state of an agent involves mapping the agent's components to the respective data structures in the AIL agent class. This includes the beliefs and plans constituting the agent's program, as well as information on the agent's context and content in the case of group, team, or organisational structures [8].

4. MODEL CHECKING

4.1 Property Specification Language

Since the AIL is intended to be a general framework with which a number of agent programming languages can interface through translators, with the ultimate goal of providing a basis for efficient model checking, the properties to be checked are specified at the MCAPL level but the semantics of the properties are provided by the implementation of the MCAPL interfaces. For agents running on an AIL-based interpreter, the semantics of the properties are specified by the AIL. The property specification language allows users to refer to agent concepts at a high level, even though JPF does model checking at the Java bytecode level.

Property specifications will be formally represented by a modal logic built on top of a (linear) temporal logic. The modalities of interest in multi-agent systems are, for example, beliefs and intentions. A typical question can be paraphrased "If agent A has the goal to achieve G and has a plan as how to bring about G, will A eventually believe G?" Such properties require temporal reasoning ('eventually') on a modal language ('A believes G'). This builds on our work on model checking AgentSpeak [3], on developing the MABLE system [16], and on formalising agents [6].

4.2 AJPF

Central to the aim of bringing uniform model checking techniques [4, 9, 7, 14] to different agent programming languages is the extension of an existing — known to be efficient — model checker. We opted for JPF because of its flexibility and extensibility. This allows us to embed the AIL classes into JPF as well as providing the means for temporal logic model checking. The embedding of the AIL classes, in turn, aims to optimise model checking for multi-agent systems by using JPF technologies to minimise the state space that needs to be checked. Property specification in our extension to JPF is possible in a meaningful, yet generic, way at the level of the MCAPL layer. In principle, model checking could be carried out without the development of the MCAPL or AIL classes and interfaces by feeding a language's Java code directly to JPF. This would, however, not allow access to any agent-specific components in a transparent way. Furthermore, in practice the memory and time required for model checking would be prohibitive (e.g., because the Java interpreters include heavy and unnecessary code such as for parsing). On the other hand, the definition of the AIL and its incorporation into JPF provides an efficient and elegant route for model checking, by mapping the key constructs of the agent language itself onto AIL concepts.

4.2.1 Java PathFinder (JPF)

JPF is an explicit state model checker for runtime-based verification of Java bytecode. Essentially, JPF implements

its own Java virtual machine on top of the host system's virtual machine[2]. The difference is that JPF's virtual machine explores *all* possible paths that may be taken in a program, continuously checking for deadlocks, violated assertions, and unhandled exceptions. (To enable this, the state space of the program has to be finite.) If JPF finds an error in a possible execution, it reports all the steps leading to that error.

Model checking in any application area is subject to the state-space explosion problem. In particular, systems involving concurrency, such as multi-agent systems, cause interleavings in the state transition system on which the model checking is carried out. Out of the box, JPF is equipped with a rich set of configuration options and abstraction mechanisms to optimise particular model checking requirements.

4.2.2 *Efficiency Issues*

The general success of model checking as a verification technique is due in great part to the various state-space reduction techniques that have been developed over the years, and made available in the most successful model checkers. Not all such techniques work well on agent-based programs, requiring agent-specific techniques to be developed (one such technique can be found in [2]). These techniques can have a major impact on the scale of implemented systems that can be effectively verified by model checking.

JPF internally employs some state-space reduction techniques. Nevertheless, we have to ensure that the state space of the transition system *relevant to model checking an agent systems* remains as small as possible. It clearly helps if we can hide as much code as possible from JPF's virtual machine or have large blocks of code be executed atomically, thereby improving the efficiency of model checking.

For our approach we have decided to restrict the relevant states that can be model checked to those agent states that are reached after a complete round of the reasoning cycle, hiding the internal state changes (i.e., the intermediate states between applications of individual semantic rules) from the model checker. We do this partly by embedding the Java-based libraries forming the AIL within JPF, in particular its new (Open Source) version, so that the basic constructs implemented in the AIL become part of the model checker. This is part of the customisations of AJPF, which is an essential component of our framework for verifying multiple agent programming languages.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have provided an overview of our new framework for the verification of multi-agent systems, incorporating agents programmed in several programming languages. This unifying approach to model checking and execution of heterogeneous agent systems will have wide benefit, as dependable systems are required in many areas of applications of agent technology.

The architecture presented in this paper is much more flexible than previous approaches to model checking for agent-based systems. Despite the greater flexibility, we have reason to believe that the system works efficiently, due to the precautions we have taken in building the architecture and the internal optimisations of AJPF, our extension of JPF.

We have developed the AIL toolkit so that new agent programming languages can easily be incorporated into the AJPF model checking architecture. Even without reprogramming a language interpreter using the AIL classes, it is possible to integrate agent programs written in a variety of languages into our verification and execution framework by interfacing directly with the MCAPL layer (with the drawback of bypassing the AJPF optimisation).

Our future plans are to carry out case studies to test the framework and to further optimise the code. The idea is to implement the case studies in different agent programming languages. Our automatic translators can then be used to translate these to the AIL platform and AJPF used to verify that the systems satisfy a specification written in our property specification language.

[2]As JPF itself is written in Java, the "host virtual machine" is the Java virtual machine used to run JPF itself.

6. REFERENCES

[1] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model Checking Rational Agents. *IEEE Intelligent Systems*, 19(5):46–52, 2004.

[2] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. State-Space Reduction Techniques in Agent Verification. In *Proc. 3rd Int. Conf. Autonomous Agents and Multiagent Systems (AAMAS)*, pages 896–903. IEEE, 2004.

[3] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying Multi-Agent Programs by Model Checking. *J. Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] L. A. Dennis, B. Farwer, R. H. Bordini, M. Fisher, and M. Wooldridge. A Common Semantic Basis for BDI Languages. In *Proc. 7th Int. Workshop on Programming Multiagent Systems (ProMAS)*, 2007.

[6] M. Fisher. Temporal Development Methods for Agent-Based Systems. *J. Autonomous Agents and Multi-Agent Systems*, 10(1):41–66, 2005.

[7] J. Hatcliff and M. B. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In *Proc. 12th Int. Conf. Concurrency Theory*, volume 2154 of *LNCS*, pages 39–58. Springer, 2001.

[8] A. Hepple, L. A. Dennis, and M. Fisher. A Common Basis for Agent Organisations in BDI Languages. In *Proc. Int. Workshop on LAnguages, methodologies and Development tools for multi-agent systemS (LADS)*, 2007.

[9] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[10] `http://javapathfinder.sourceforge.net`.

[11] M. Kacprzak, A. Lomuscio, and W. Penczek. Verification of Multiagent Systems via Unbounded Model Checking. In *Proc. 3rd Int. Conf. Autonomous Agents and Multiagent Systems (AAMAS)*, pages 638–645. IEEE CS Press, 2004.

[12] F. Raimondi and A. Lomuscio. Automatic Verification of Multi-agent Systems by Model Checking Ordered Binary Decision Diagrams. *J. Applied Logic*, 5(2):235–251, 2007.

[13] A. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996.

[14] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an Extensible and Highly-Modular Software Model Checking Framework. In *Proc. ESEC / SIGSOFT FSE*, pages 267–276, 2003.

[15] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.x

[16] M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model Checking for Multiagent Systems: The MABLE Language and its Applications. *Int. J. Artificial Intelligence Tools*, 15(2):195–225, 2006.