Automated Verification of Multi-Agent Programs

Rafael H. Bordini*, Louise A. Dennis†, Berndt Farwer*, Michael Fisher†

* Deptartment of Computer Science, Durham University, UK
Email: {R.Bordini,berndt.farwer}@durham.ac.uk

† Dept. of Computer Science, University of Liverpool, UK
Email: {l.a.dennis,mfisher}@liverpool.ac.uk

***Abstract*—In this paper, we show that the flexible model-checking of multi-agent systems, implemented using agent-oriented programming languages, is viable thus paving the way for the construction of verifiably correct applications of autonomous agents and multi-agent systems. Model checking experiments were carried out on AJPF (Agent JPF), our extension of Java PathFinder that incorporates the Agent Infrastructure Layer, our unifying framework for agent programming languages. In our approach, properties are specified in a temporal language extended with (shallow) agent-related modalities. The framework then allows the verification of programs written in a variety of agent programming languages, thus removing the need for individual languages to implement their own verification framework. It even allows the verification of multi-agent systems comprised of agents developed in a variety of different (agent) programming languages. As an example, we also provide model checking results for the verification of a multi-agent system implementing a well-known task sharing protocol.**

I. Introduction

We view an agent as an *autonomous* computational entity making its own decisions about what activities to pursue. Often, this involves having explicit representation of *goals* to achieve and being able to communicate with other agents in order to accomplish these goals [1]. *Rational agents* make such decisions in a rational and explainable way and, since agents are autonomous, understanding *why* an agent chooses a particular course of action is vital. Therefore, when designing or analysing such agents, it is vital to consider not just what they do but *why* they do it.

The ability of agents to act independently, to react to unexpected situations, and to co-operate with other agents, has made them a popular choice for developing software in a number of areas. At one extreme there are agents that are used to search the INTERNET, navigating autonomously in order to retrieve information; these are relatively lightweight agents, with few goals but significant domain-specific knowledge. At the other end of the spectrum, there are agents developed for independent process control in unpredictable environments. This second form of agent is often constructed using complex software architectures, and they have been applied in areas such as autonomous spacecraft control [2], health care [3], and process control [4], [5]. Clearly, these are areas for which we often demand both *dependability* and *security*.

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, formal verification techniques tailored specifically for agent-based systems is an area that is now attracting a great deal of attention [6], [7]. While program verification is well advanced, for example Java verification using Java PathFinder [8], [9], the 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 must verify not only *what* the agent does, but *why* it chose that particular course of action, what it *believed* that made it choose to act in a particular way, and what its *intentions* were in doing so. This often leads to the need to extend the temporal basis of verification with notions such as agent *belief* and agent *intention*, both of which are typically characterised as *modal* dimensions.

However, there are now very *many* agent programming languages and agent platforms (often provided as extensions of Java). Rather than providing an approach in which the complex logical properties of systems using just *one* particular agent approach can be verified, we have developed a flexible framework allowing the verification of a wide range of agent-based programs, produced using various high-level agent programming languages. As this is a complex endeavour, we have built up its basis over a number of years. Thus, the modelling and verification framework implemented here is based on several earlier results:

- in [10], we provided an overview of the proposed systems architecture;
- in [11], we provided the formal basis for the libraries into which various agent programming languages can be translated;
- in [12], we analysed the operational semantics of such agent languages using a formal tool (Maude); and
- in [13], we showed that *heterogeneous* multi-agent programs (with agents implemented in different languages) can be translated into in our framework.

In this paper, we show that the proposed framework not only works in practice but provides a viable approach to developing *verifiable multi-agent programs* across several agent programming languages. We here present, for the first time, the results of using our framework to model check a multi-agent system where agents use a well-known protocol for task sharing among agents.

The structure of the paper is as follows. In Section II, we discuss agent programming languages, and introduce some key issues concerning the verification of agent-based systems. In Section III, we provide an overview of our Agent Infrastructure Layer (AIL) followed, in Section IV, by some details on 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) of Java PathFinder (JPF) in Section V. In Section VI, we present examples of the verification of multi-agent systems, and finally, in Section VII, we provide concluding remarks.

II. Background

A. BDI Programming Languages

The key reason why an agent-based approach is advantageous in the modelling and programming of autonomous systems is that it permits the clear and concise representation not only of *what* the autonomous components within the system do, but also *why* they do it. This allows us to abstract away from the low-level control aspects and to concentrate on the key feature of autonomy, namely the goals each component has and the choices it makes. Thus, in modelling a system in terms of agents, we often describe each agent's *beliefs* and *goals* (also called desires), which in turn determine the agent's *intentions*. Such agents then make decisions about what actions to perform, given their current beliefs, goals, and intentions. This kind of approach has been popularised through the influential BDI (Belief-Desire-Intention) model of agency [4] and, although this representation of behaviour using *mental* notions is initially unusual, it has several benefits. The first is that, ideally, it abstracts away from low-level issues: we simply present some goal that we wish to be achieved, and we expect the agent to act in what we would consider a reasonable, or *rational*, way given such a goal. Secondly, because we are used to understanding and predicting the behaviour of rational (human) agents, the behaviour of autonomous software should be relatively easy for humans to understand and predict too. The modelling of complex systems in terms of rational agents captured within the BDI approach has been very successful, for example [14], [15].

While agent-based systems were originally programmed using standard programming languages, such as Java, there is now a recognition that such languages, on their own, are not sufficient to concisely represent the key aspects of agents, in particular the motivation the agent has for undertaking some action. As a consequence, a range of agent programming languages and platforms are being seriously developed, for example 3APL, *Jason*, Jadex, and METATEM; see [14] for an overview of some such languages. The nature of the area is such that it is unlikely that one single agent language will be used in all the areas of application for multi-agent systems. As a result as well as providing a generic approach to the verification of multi-agent systems written in a single agent programming language, our approach is also an important step towards code reusability crossing the borders of single (agent) programming languages.

B. Verification via Model Checking

The growing complexity of software systems combined with their increasing use in security, data protection, health, etc. make formal verification of key aspects of such systems vital. Following this trend, there is an increasing need for verification of agent-based systems, especially where *dependable* agent-based applications need to be developed.

Formal verification has traditionally been approached via mathematical theorem proving, usually undertaken on a model of the real system and requiring a high degree of mathematical ability on the part of the user operating the proof tools. However formal verification can also be achieved through model checking [16], [17], [18] an area of research that has produced impressive results in recent years. Model checking is increasingly used in industry, since the process is fully automatic once a formal model of the system is obtained and a property has been specified.

More specifically, model checking is a technique whereby a finite description of a system is analysed with respect to a property in order to ascertain whether *all* possible executions of the system satisfy this property. Different application areas may require specialised property specification languages, but *temporal logics* have proved to provide a sufficiently generic basis. Temporal logic is a mathematically strict formalism applicable in a variety of theoretical and practical aspects of Computer Science and Artificial Intelligence, including its use as a property specification language for aspects of hardware and software verification [19], [20].

As mentioned above, in our work it is vital to verify not only the behaviour that the agent system has, but to verify *why* the agents are undertaking certain courses of action. Thus, the temporal basis of model checking must be extended with notions such as agent *belief* and agent *intention*, both of which are characterised as *intensional modal operators* [21]. While the temporal component captures the *dynamic* nature of agent behaviour, the modal components capture 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 [22], [23], [24], [25]

III. Agent Infrastructure Layer

Previous approaches to model checking agent-based systems have mostly been specific to one particular programming language (e.g., [24]), while in practice there is a wide variety of relevant agent programming languages. Such approaches to model checking agent programs have also relied on encoding beliefs, goals, etc., within the state of the the model checker's transition system, e.g. the JPF [8] or Spin [17] state machine. This is a complex task, and one that would need to be (at least partly) redone to allow model checking of different programming languages. Consequently, the need for a unifying framework arises. Drawing from previous work on verifying

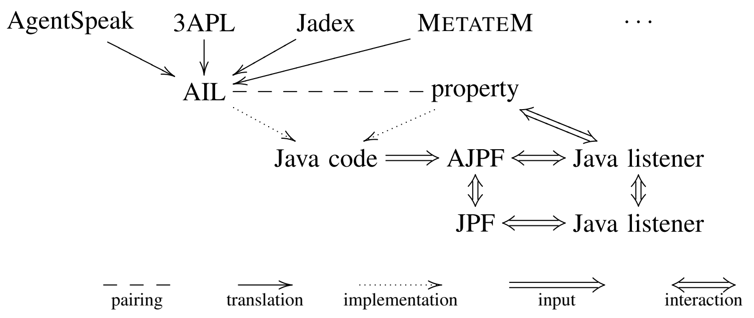

Fig. 1. Our Approach [10]

agent-based systems developed in AgentSpeak [22], [26], [24], we have developed a framework for bringing a large part of the verification-related aspects of that work together. The Agent Infrastructure Layer (AIL) [11] encompasses the main concepts from a wide range of agent programming languages. Technically speaking, it is a *toolkit* collecting together Java classes that:

(i) facilitates implementation of interpreters for various agent programming languages;
(ii) contains adaptable, clear semantics; and
(iii) can be verified through AJPF, an extended version of the open source Java model checker JPF [9].

AJPF is a customisation of JPF that was optimised for AIL-based interpreters; see Section V. Figure 1 shows the overall architecture of our new approach [10].

The AIL is further extended with the *MCAPL interface*[1], which is required for actual model checking. This interface also 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, using AJPF. However these languages will not benefit from the efficiency improvements that the optimised AIL classes provide.

Within the AIL we assume that agents, written in any agent programming language, all possess a *reasoning cycle* consisting of a number (possibly only one) of stages (a reasoning cycle can often be broken down in various identifiable stages that help formalisation and understanding). Each stage is typically formalised as a disjunction of semantic rules which define how an agent's state may change during the execution of that stage. The combined rules of the various 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 may already exist in the toolkit) and the implementation of a reasoning cycle, by organising the rules into (the stages of) such a cycle. In this way, we have implemented, for example, both GOAL [27] and SAAPL (Simple Abstract Agent Programming Language) [28] interpreters [13], following their respective operational semantics. The implementations of these interpreters make use of the AIL operations together with some additional classes specifically added to faithfully reproduce the semantics of those languages.

[1]MCAPL stands for "Model Checking Agent Programming Languages".

Initially, we developed the AIL by implementing an operational semantics introduced in [11]. This turned out to be too inflexible to accommodate the reasoning cycle of some agent languages. Instead, we now identify key *operations* that many (BDI-)languages use and treat these operations as part of the *AIL toolkit*. The rules in [11] (together with some obvious alternatives) then become a part of this toolkit. For any given language, it may be sufficient to use only a selection of these rules (when developing its own AIL-based interpreter), or it may be necessary to add custom rules built from the basic operations. These operations and rules have formal semantics and are implemented as Java classes or methods.

An agent originally programmed in some agent programming language 'APL' and running in an AIL-based interpreter uses the 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. 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 potentially relevant for verification. This functionality is obtained simply through the use of AIL for the development of the interpreter; AIL also makes it easier to develop an interpreter for a programming language than doing it "from scratch" in Java. Figure 2 provides a diagrammatic representation of the AIL within the AJPF model checking architecture which is discussed in more detail in Section V-B.

A typical rule used in the operational semantics of an agent programming language is one for adding a belief to the belief base:

$$\frac{}{\langle BB, +b : I, \mathbf{S} \rangle \rightarrow \langle BB \cup \{b\}, I, \mathbf{S'} \rangle}$$

In this rule, we represent the belief base (a set of beliefs) component of the agent state, as BB, and also a stack (the *intention*, I) of things to do[2], with $+b$ (add belief b) on the top of this stack. The rule represents the change made to the agent state by processing the $+b$ at the top of the stack. The

[2]For simplicity of presentation, we omit other parts of the agent state from the statement of the rule.

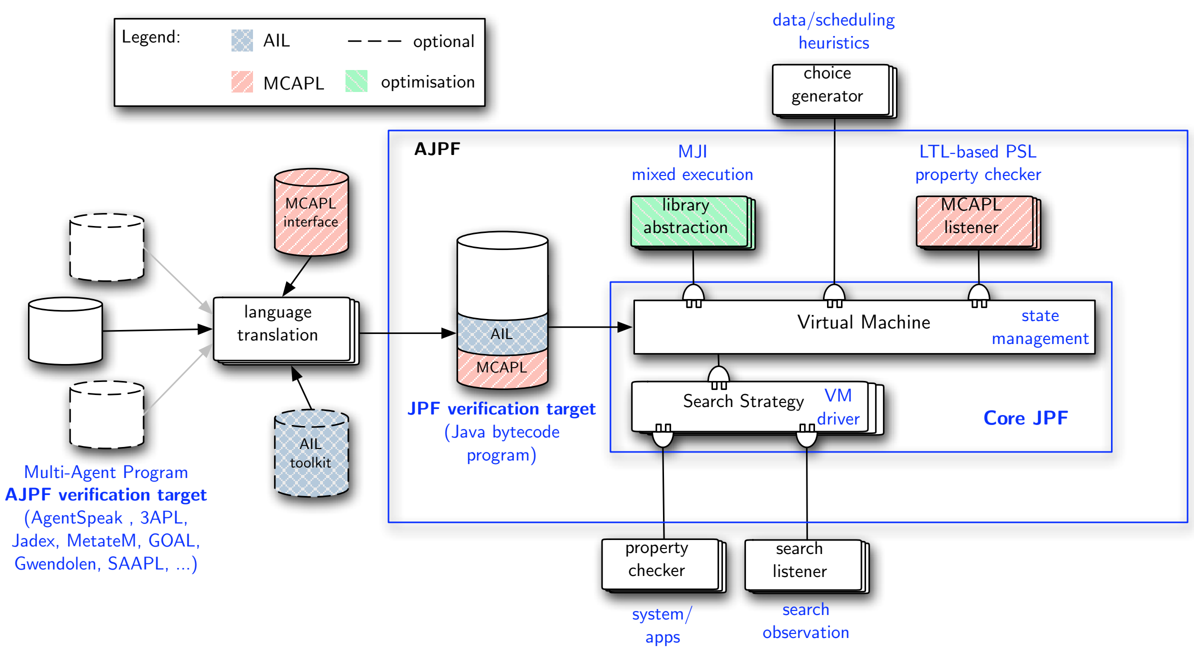

Fig. 2. Overview of the AJPF architecture

stage of the reasoning cycle is also updated by the rule (from **S** to **S'**, according to the semantics of the language being interpreted). This rule is available in the AIL Toolkit for use in the development of language interpreters. When it is assigned to a stage of a reasoning cycle, **S** and **S'** are instantiated appropriately. The data structures used in the above example (i.e., BB, $+b$, and I) are all part of the default AIL agent class, making the rule usable without modification for most language implementations.

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) [9]. The JPF model checker is extensible and configurable, which allows us to optimise its performance for AIL-based systems.

IV. Developing BDI-based Interpreters using AIL

Among agent programming languages, some of the most widely used rely on the *belief-desire-intention* (BDI) paradigm. For this reason, we have chosen to focus on some major BDI languages as primary case study candidates for the use of AIL; in particular, we started from 3APL [29] and the variant of AgentSpeak [30] encapsulated by *Jason* [31]. However, our approach does not exclude other languages, even those based on completely different paradigms.

Another prerequisite of our endeavour was to include languages that have practical relevance. That is, we did not want to restrict ourselves to (abstract) programming languages that cannot be considered for serious software development projects. We have already shown that "leaner" languages which have simple semantics will mostly embed quickly and straightforwardly into our framework [13].

A. BDI Foundations of AIL

The architecture presented in this paper is based upon our study of common concepts and structures appearing in the operational semantics of various BDI programming languages. In [11], we tackled issues in the treatment of events, goals, intentions, and other components central to the design of these languages.

The AIL semantics include appropriate data structures and operations for beliefs, plans, constraints, messages, content, and context of an agent. Furthermore, the semantics formally define a number of rules that are associated with different stages of a typical agent reasoning cycle, such as rules for the selection of an intention, generating the set of applicable plans, updating the components of an agent's state, adding beliefs, executing actions, dropping goals, handling messages (i.e., inter-agent communication) and perception (of the environment state), etc.

We introduced the term "deed" in [11] as a way to refer to the various types of formula one can typically have in the body of plans. A deed stack is the core of the AIL's most complex data structure which represents an *intention*. BDI languages use intentions to store the *intended means* for achieving goals. Intention structures may also maintain information about the (sub-)goals they are intended to achieve or the event that triggered them. In the AIL, new events are associated with an empty deed (ϵ).

The AIL's basic plan data structure associates a trigger with a guard and a deed stack. The AIL includes operations for using such plans to modify intentions in a way which supports the use of plans in many BDI languages. Mapping plans from common agent programming languages onto an AIL representation has been, in our experience, straightforward.

By way of example, consider the following AIL plan for cleaning rooms.

trigger	guard	body
`+!clean()`	`dirty(Room)`	`+!Goto(Room)` `+!Vacuum(Room)`

This plan has a trigger which is a goal to clean a room, a guard which is a requirement the agent believes the room to be dirty, and a body to be executed which consists of two new goals to be adopted: first to go to the dirty room; second to vacuum that room.

The following shows the operation of AIL's default plan execution on an intention, given the plan above.

trigger	deed
`+!clean()`	ϵ

$\longrightarrow$

trigger	deed
`+!clean()`	`+!Goto(Room)`
`+!clean()`	`+!Vacuum(Room)`

The plan's trigger matched the top event of this intention[3]. If the guard is a logical consequence of the agent's beliefs, the AIL removes the top row of the intention and replaces it with the two rows representing the body of this plan. Semantic rules can then be used, as appropriate to the language, to move the top deed (the goal to go to the dirty room) to an event stack and to trigger further plan matching to achieve that sub-goal.

B. Translating Languages

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 default semantics for 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 data structures. In particular, the initial state of an agent has to be translated into an AIL agent state.

Once an agent (in a supported programming language) has been translated so that it uses the AIL, not only can it be verified, but also executed, given an appropriate execution environment, together with any other agents, possibly programmed in other languages translated into the AIL. This also makes it possible to verify such heterogeneous multi-agent systems.

The AIL data structures and rules facilitate the implementation of language interpreters. These building blocks can be used to re-create the operational semantic rules of the target language. Using these Java classes and their associated methods makes programming the language interpreter much easier than doing it "from scratch" using Java. The AIL provides all the infrastructure that is needed for a full implementation of an agent programming language (and associated agent execution platform) so that the only major task required to implement a language is the combination of AIL operations to form the specific semantic rules. Taking into consideration that those operations include all the basic querying and updating of agent state components, the rules are relatively easily constructed.

An interpreter written using the AIL toolkit provides its own custom sub-class of the AIL agent class which is suitable for the implicit data-structures of the language under consideration. It is our experience that many of the AIL data structures can be used without modification for this purpose. It is, of course, then also necessary to provide a parser for that given language in order to generate the appropriate instance of the respective agent class. This way, from the user's perspective, no effort is required to prepare the program to be verified since the original program code is directly fed into AJPF. After this is done, it will still be necessary for the user to define the properties to be (model) checked.

C. Implementing the AIL

Based upon the AIL semantics presented in [11], we have implemented Java classes forming the AIL toolkit. The toolkit has classes for all the AIL data structures, including the agent class which embodies the state of an AIL agent. This agent class assumes the provision of a language-specific reasoning cycle constructed from (custom) operational semantic rules; it is this agent class which an interpreter for a specific language is expected to subclass. All the semantic rules presented in [11] are available for use in specific reasoning cycles but it is also possible to construct new rules and use them in a particular reasoning cycle.

In addition to the AIL, there is also the MCAPL interface (see Figure 2) provided for model checking a multi-agent program against a property specification written in the Property Specification Language (PSL) introduced later in Section V-A.

The MCAPL layer requires that, for any given agent programming language, two interfaces are implemented: one for individual agents and another for the overall multi-agent system. This software layer provides a MCAPL controller which requests a list of agents from the multi-agent system and encapsulates these in a special object which alternately calls one reasoning step of each agent, anticipated to be one full run of the reasoning cycle and then checks it against the specification by calling, for instance, methods that implement belief checking as defined by the specific language. Properties are also checked when JPF detects that an "end state" is reached (this could indicate a cycle in the states of a run as well as program termination).

As shown in Figure 2, the combination of the translated agent program(s) with the AIL and MCAPL machinery (including the translated property specification) constitute the JPF verification target. The original program(s) with the original property specification are the AJPF verification target; they are fed into the appropriate translators available as part of our framework.

[3]In fact the AIL's default plan execution mechanism allows for matching of the deed stack prefix as well, as required in a 3APL interpreter for example, but we omit this for simplicity.

V. Model Checking

A. Property Specification Language

Since we aim to provide a general framework for model checking with which various agent languages can interface (whether AIL-based or not), the properties to be checked are specified at the MCAPL level. For agents running on an AIL-based interpreter, the semantics of the properties is already specified as part of the AIL toolkit itself. The property specification language (PSL) allows users to refer to agent concepts at a high level, even though JPF carries out model checking at the Java bytecode level.

The MCAPL layer allows agents to be model checked using essentially the same property specification language without using the AIL's data structures or implementing an AIL interpreter for that language. To use the MCAPL interface, the language must implement the MCAPL interface classes appropriately. This implementation defines the required modalities of the property specification language. For instance, agents implementing the MCAPL agent interface must provide a method which succeeds when the agent *believes* the given parameter (represented as a "formula") is true. Typically, agent programming languages do not fully implement logics of belief, so the property specification language contains no provision for nesting the modal operators and assumes that the modalities are very simple — however, this does not preclude users, when implementing the MCAPL interface, from developing such a logic based on their agent state. The implementation of the modalities defines their semantics (e.g., for *belief*) in that specific 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 AIL-based interpreters are constructed in a way that makes the AIL semantics of the properties consistent with the language's individual semantics for those modalities (otherwise they cannot use the AIL implementation and will need to override it using the MCAPL interface).

The property specification language is based on a fragment of LTL (Linear Temporal Logic) [20] containing $\neg, \vee, \wedge, \mathbf{U}, \mathbf{R}$ and enriched by modalities such as $\mathbf{B}, \mathbf{G}$ ('ag believes' and 'ag has goal'). The temporal operators $\lozenge$ *(eventually)* and $\square$ *(always)* are derived from the above, as usual.

Properties specified through the MCAPL interface can be checked using the JPF model checker. At present we can verify properties of programs using properties of the form:

$$\begin{array}{rl} ag ::= & \text{"agent constant"} \\ f ::= & \text{"ground first order atomic formula"} \\ \phi ::= & \mathbf{B}(ag,f) \mid \mathbf{G}(ag,f) \mid \mathbf{A}(ag,f) \mid \mathbf{I}(ag,f) \mid \mathbf{P}(f) \\ & \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi\,\mathbf{U}\,\phi \mid \phi\,\mathbf{R}\,\phi \end{array}$$

Consider a program, P, describing a multi-agent system and let MAS be the state of the multi-agent system at one point in the run of P. Consider an agent, $ag \in MAS$, at this point in the program execution. Then

$$MAS \models_{MC} \mathbf{B}(ag,f) \text{ iff } f \in ag_{BB}$$

where ag_{BB} is the belief base of the agent, ag. An agent's belief base consists of a set of ground formulae[4]. Similarly, the interpretation of $\mathbf{G}(ag,f)$ is given as[5]

$$MAS \models_{MC} \mathbf{G}(ag,f) \text{ iff } !_a f \in ag_G$$

where ag_G is the set of goal commitment events stored in the intentions of agent ag. $MAS \models_{MC} \mathbf{A}(ag,f)$ iff the last action changing the environment was action f taken by agent ag. $MAS \models_{MC} \mathbf{I}(ag,f)$ iff $!_a f \in ag_G$ and there is an intended means for f (i.e., it is not associated with the "no plan yet" element ϵ). $MAS \models_{MC} \mathbf{P}(f)$ iff f is a percept that holds true in the environment.

The other operators in the MCAPL property specification language have standard LTL semantics and are implemented by the MCAPL interface:

$$\begin{array}{rcl} MAS \models_{MC} \varphi \wedge \psi & \text{iff} & MAS \models_{MC} \varphi \text{ and } MAS \models_{MC} \psi \\ MAS \models_{MC} \varphi \vee \psi & \text{iff} & MAS \models_{MC} \varphi \text{ or } MAS \models_{MC} \psi \\ MAS \models_{MC} \neg\phi & \text{iff} & MAS \not\models_{MC} \phi. \end{array}$$

The temporal formulæ apply to runs of the programs in the JPF model checker. A run consists of a (possibly infinite) sequence of program states MAS_i, $i \geq 0$ where MAS_0 is the initial state of the program (note, however, that for model checking the number of different states in any run is assumed to be finite). Let P be a multi-agent program, then $P \models_{MC} \varphi\,\mathbf{U}\,\psi$ holds iff in all runs of the P there exists a state MAS_j such that $MAS_i \models_{MC} \varphi$ for all $0 \leq i < j$ and $MAS_j \models_{MC} \psi$. Similarly, $P \models_{MC} \varphi\,\mathbf{R}\,\psi$ holds iff either $MAS_i \models_{MC} \varphi$ for all i or there exists MAS_j such that $MAS_i \models_{MC} \varphi$ for all $i \in \{0, \ldots, j\}$ and $MAS_j \models_{MC} \varphi \wedge \psi$. The temporal operators $\lozenge$ (eventually) and $\square$ (always) are derivable from $\mathbf{U}$ and $\mathbf{R}$ [20]. It should be noted that this implementation represents a significant addition to JPF which, since it was released as *Open Source*, no longer supports LTL model checking.

B. AJPF

Central to the aim of bringing uniform model checking techniques 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, and because most agent platforms are based on Java and allow legacy Java code to be used by the agents. We have embedded the AIL classes into JPF, and we have also provided 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 techniques that help minimise the state space that needs to be checked. For instance, our implementation of the classes makes use of a

[4]We intend to extend this to include Prolog-style reasoning on the belief base using *belief rules* e.g., as in 3APL and *Jason*

[5]NB: The notation $!g$ is typical for a goal in BDI languages. We use $!_a g$ specifically for *achievement* goals — used to state that the agent wishes the associated propositions to hold in the future — and $!_p g$ for *perform* goals — which encapsulate a sequence of deeds but do not relate to a particular belief the agent expects to acquire after they are executed, as is the case with achievement goals.

technique that allows the execution of certain methods to be moved from the (slow) virtual machine provided by JPF to the (faster) host virtual machine when appropriate — we discuss this further below.

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 the Java code of the interpreter of an agent language directly to JPF. This would, however, not allow access to any agent-specific components in a transparent way (besides being heavily language dependent). 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). Before we discuss our JPF extension in more detail, let us first consider some important features of JPF.

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[6]. The JPF Java virtual machine is changed to explore *all* possible paths that may be taken in a program's execution, continuously checking for deadlocks, violated assertions, and unhandled exceptions. (To enable this, the state space of the program has to be finite, of course.) If JPF finds an error in one of the possible executions, it immediately reports to the user all the steps leading to that error.

Model checking in any application area is subject to scalability problems, i.e., with the linear growth of the size of the system to be checked, there is an exponential growth of the state space that has to be explored. In particular, systems involving concurrency, such as multi-agent systems, lead to interleavings in the transition system on which model checking is to be carried out. To cope with the time and memory requirements of large systems, the model checker has to provide techniques for dealing with large memory structures representing system states, and ideally reducing the state space while still ensuring that all possible paths of the system (relevant for a given property) are checked. JPF can be adapted in several ways to suit different applications and should be viewed as an extensible framework for Java bytecode model checking. *Out of the box*, JPF is equipped with a number of settings and abstraction techniques that can be configured to suit the requirements of specific model-checking exercises.

2) Properties and JPF Listeners: The latest version of JPF (4.1) does not include a logic-based property-specification language. Since, in industry, JPF is used mainly for debugging (rather than verification), by default JPF only checks for inline Java assertions, deadlocks, and Java exceptions. However, JPF allows the user to program *listeners* that monitor certain aspects of the Java execution in JPF's virtual machine.

We use AJPF listeners to implement model checking of properties specified in our property specification language. Listeners can be used to monitor various types of events in the execution environment, expanding the internal model checking mechanism of JPF. To check a specific property given in our property specification language, the property is negated and then translated into a property automaton using standard authomata-theoretic approaches [32], [33]. This automaton maintains a set of valid current states. The automaton object is part of the the MCAPL layer (i.e., the code is run in the JPF virtual machine) and, at the end of each round of the reasoning cycle in any of the agents, the automaton's current state set is updated and checked for emptiness (which implies the (negated) property has been satisfied) or if it has entered a loop consisting only of accepting states (which implies the property is violated). By running the property automaton in the JPF virtual machine alongside the program, we create an automaton in JPF that represents the product automaton of the multi-agent program and the property to be checked. Each time the property automaton is updated, the JPF listener is notified of the current status of the automaton and generates a violation or prunes the search space as appropriate. The listener also detects when the program/property automata pair running in the MCAPL layer has reached an end state and checks that the property automaton is not in any accepting state.

3) Efficiency Issues: The success of model checking is due in great part to various state-space reduction techniques that are available in the most successful model checkers. Not all such techniques work well on agent programs, requiring agent-specific techniques to be developed (an example of such a technique can be found in [26]).

JPF employs various state-space reduction techniques, for example on-the-fly partial-order reduction (i.e., combining instruction sequences that only have effects inside a single thread and executing them as a single transition). Nevertheless, we have to ensure that the state space *relevant to an agent system* remains as small as possible. We need to ensure that only relevant backtracking points are stored, thereby limiting the state space and improving the efficiency of model checking.

In our approach, we have decided to restrict the states that are generated during model checked to the states that are reached after a complete round of an agent's reasoning cycle. The effect is a dramatic reduction in the size of the state space, but as a consequence users need to ensure that the system properties they specify are not sensitive to intermediate changes in the agent state. Executing the initialisation of the agents and of the MAS *atomically*, reduces typical verification times by 30%–40%. The substantial speed-up happens because this portion of the code is executed many times as JPF backtracks. Further use of atomic sections within the reasoning cycle dramatically improves efficiency (see Section VI-A).

VI. Case Studies

In this section we summarise the results of two model checking experiments: one consisting of a simple agent program and another of a system of multiple communicating

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

TABLE I
MODEL CHECKING STATISTICS

$\Diamond_{ag_1}$pickup	JPF	AJPF
elapsed time:	0:07:03	0:00:04
states:	visited=11080, backtracked=22223	visited=11, backtracked=34
choice generators:	thread=11145	thread=25
heap:	gc=28161	gc=65
instructions:	235599025	1051314
max memory:	59MB	26MB

agents. All the following examples are implemented in a prototype agent language [34] that has been useful in various stages of our project. The details of the language are unimportant and we use standard BDI-style syntax — a plan is a triple $e : g \leftarrow ds$ consisting of a triggering event e, a guard g, and a body ds (a stack of deeds). As common in BDI-style languages, an event $+b$ means belief b has just been acquired, $+!_a g$ denotes the adoption of a new achievement goal g, and similarly $+!_p g$ for a perform (rather than achievement) goal g being adopted[7]; in a plan body, a is used to represent an action that the agent will perform when executing the plan. This prototype language has an interpreter written in AIL and so verification can be undertaken with all the optimisations of AJPF in place.

A. Simple Agent Systems

We show how using AJPF to model check a simple (single) agent program improves the performance of JPF by juxtaposing the model checking runs and their statistics. The improved performance is a result of the use of optimised code for the underlying AIL classes. This optimisation is, to a large extent, due to the introduction of atomic sections reducing the number of possible interleavings[8] in the non-optimised version.

In the following example, the environment provides some shared resources, the availability of which are perceived by the agent. The agent updates its beliefs about the world based upon its perception. The agent beliefs, updated with such percepts, are used to achieve the goal of picking up a block by using two plans. The first plan establishes a new goal to be adopted whenever the agent perceives a new block. The second plan actually tries to get hold of the block.

$$+block_i : \top \;\leftarrow\; +!_a pickup_i$$
$$+!_a pickup_i : block_i \wedge empty \;\leftarrow\; pickup_i; -empty; +busy$$

In a similar example, the property that the initial goal of picking up an object is eventually achieved by the single agent is verified by AJPF in 4 seconds, while JPF needed more than 7 minutes (on the same machine). While the number of classes and methods remain the same, the optimised AJPF requires less than half of the memory, only about 1/1000 of visited and backtracked states as well as choice generators, 1/200 of instructions, and 1/50 of garbage collections (gc) within JPF's heap, resulting in a speed-up of more than 150× (see Table I).

As expected, a two-agent version of this example increases the state space significantly due to concurrency.

B. Multi-Agent System with Communication

The system studied in this section comprises two agents, ag_1 and ag_2, and an object to be picked up. ag_1 asks ag_2 to pick the object up, since only ag_2 has the know-how (i.e., a plan) to do so.

Agent ag_1 has the initial achievement goal, $!_a pickup$, and one plan, as follows.

$$+!_a pickup : \top \leftarrow \mathbf{send}(ag_2, (\mathbf{achieve}, pickup)); \; \texttt{wait}$$

This plan tells agent ag_2 to pick up the object and then wait.

Agent ag_2 has the following two plans:

$$+!_a pickup : \top \;\leftarrow\; pickup$$
$$+\mathbf{receive}(ag_1, (\mathbf{achieve}, Goal)) : \top \;\leftarrow\; +!_a Goal$$

We are able to show, for instance, the achievement of an agent's initial goal, e.g., $\Diamond(\mathbf{B}(ag_1, pickup))$ ("eventually, ag_1 believes that the object is picked up")

We have also used our approach to check the main properties of interest for a variation of the contract net scenario we used in [13]. This has been carried out for two- and three-agent versions of the protocol.

The agents' plans for achieving a goal g, either by performing an appropriate *action* a, or committing to performing a *call for proposals* (cfp) are:

$$+!_a g : cando(g) \;\leftarrow\; a$$
$$+!_a g : \neg cando(g) \;\leftarrow\; +!_p cfp(g)$$

Similar plans exist for a second goal g'. The contract net protocol assumes a message semantics consisting of a *performative* and a *ground formula*. Two performatives are used: $\mathbf{perform}$ instructs an agent to perform an action (or more generally to achieve a goal) and $\mathbf{tell}$ instructs an agent to update its belief base. The implementation of the protocol includes a plan asking an agent to respond to a $\mathbf{perform}$ request, together with a number of plans for responding and how to act if an agent has a proposal or is awarded a contract (see Figure 3).

One of the properties that can be checked for this program is that eventually an agent ag_i achieves the initial goal g — formally $\Diamond(\mathbf{B}(ag_i, g))$.

[7]Achievement and perform goals are as described in Section V-A.

[8]Note that even in the single agent case there are 2 execution threads, one for the agent itself and another representing the agent's environment.

$$\begin{aligned}
+!_p\, cfp(T) : ag(A) \land my\_name(N) \land \sim \mathbf{send}(A, (\mathbf{perform}, respond(T, N))) &\leftarrow \mathbf{send}(A, (\mathbf{perform}, respond(T, N)); \texttt{wait}) \\
+!_p\, cfp(T) : proposal(T, A) &\leftarrow \texttt{wait} \\
+!_p\, respond(T, A) : cando(T) \land my\_name(N) &\leftarrow \mathbf{send}(A, (\mathbf{tell}, proposal(T, N))) \\
+!_p\, respond(T, A) : \neg cando(T) \land my\_name(N) &\leftarrow \mathbf{send}(A, (\mathbf{tell}, sorry(T, N))) \\
+proposal(T, A) : \top &\leftarrow \mathbf{send}(A, (\mathbf{tell}, award(T))) \\
+award(T) : \top &\leftarrow +!_a\, T
\end{aligned}$$

Fig. 3. Contract-Net Example

VII. Conclusions and Future Work

In this paper, we have presented an overview of our framework for verifying multi-agent systems where agents can be programmed in a variey of agent-oriented programming languages. This unifying approach to model checking and execution of heterogeneous agent systems will have significant benefits, 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 it 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 so that new agent programming languages can easily be incorporated into our framework. Even without re-programming 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. However, in the latter case, the user will not be able to take advantage of the AJPF optimisations which we have shown to make model checking of AIL-based systems significantly more efficient. Our future work aims to utilise more sophisticated aspects of JPF, such as its "Model java Interface" [35], to improve performance further.

In [12], we describe a prototype in Maude [36] of the AIL and an implementation of AgentSpeak. In future work, we also aim to use those prototypes to help proving correctness of the automated translations into the AIL classes.

We further plan to carry out more case studies so as to test the framework in more realistic scenarios, and to further optimise the AIL code and AJPF. The areas in which we seek to tackle case studies include: (i) autonomous spacecraft control; (ii) network security; (iii) negotiation/cooperation in commercial/industrial applications of multi-agent systems; and (iv) autonomous agents for pervasive computing. 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 can be used to verify that the systems satisfy specifications written in our property specification language. These case studies will be generated through our close collaboration with organisations such as NASA and industrial partners.

Acknowledgements

The work reported in this paper has been supported through EPSRC grants EP/D054788 (Durham) and EP/D052548 (Liverpool).

References

[1] M. Wooldridge and N. R. Jennings, "Intelligent Agents: Theory and Practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[2] N. Muscettola, P. P. Nayak, B. Pell, and B. Williams, "Remote Agent: To Boldly Go Where No AI System Has Gone Before," *Artificial Intelligence*, vol. 103, no. 1-2, pp. 5–48, 1998.

[3] A. Moreno and C. Garbay, "Software Agents in Health Care," *Artificial Intelligence in Medicine*, vol. 27, no. 3, pp. 229–232, 2003.

[4] A. S. Rao and M. Georgeff, "BDI Agents: From Theory to Practice," in *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, San Francisco, CA, Jun. 1995, pp. 312–319.

[5] N. R. Jennings and M. Wooldridge, "Applications of agent technology," in *Agent Technology: Foundations, Applications, and Markets*. Springer-Verlag, Heidelberg, 1998.

[6] M. Greaves, V. Stavridou-Coleman, and R. Laddaga, "Dependable Agent Systems (Editorial)," *IEEE Intelligent Systems*, vol. 19, no. 5, pp. 20–23, 2004.

[7] M. Fisher, M. P. Singh, D. F. Spears, and M. Wooldridge, "Logic-Based Agent Verification (Editorial)," *Journal of Applied Logic*, vol. 5, no. 2, pp. 193–195, 2007.

[8] "Java PathFinder," `http://javapathfinder.sourceforge.net`.

[9] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model Checking Programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[10] L. A. Dennis, B. Farwer, R. H. Bordini, and M. Fisher, "A Flexible Framework for Verifying Agent Programs (Short Paper)," in *Proc. 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*. ACM, 2008, to Appear.

[11] L. A. Dennis, B. Farwer, R. H. Bordini, M. Fisher, and M. Wooldridge, "A Common Semantic Basis for BDI Languages," in *Proc. 7th International Workshop on Programming Multiagent Systems (ProMAS)*, ser. Lecture Notes in Artificial Intelligence. Springer Verlag, 2007 (to appear).

[12] B. Farwer and L. A. Dennis, "Translating into an Intermediate Agent Layer: A Prototype in Maude," in *Proc. International Workshop on Concurrency, Specification and Programming (CS&P)*, Lagow, Poland, September 2007.

[13] L. A. Dennis and M. Fisher, "Programming Verifiable Heterogeneous Agent Systems," in *Proc. 6th International Workshop on Programming Multiagent Systems (ProMAS)*, 2008, (To appear).

[14] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, Eds., *Multi-Agent Programming: Languages, Platforms and Applications*. Springer-Verlag, 2005.

[15] M. Sierhuis, J. M. Bradshaw, A. Acquisti, R. V. Hoof, R. Jeffers, and A. Uszok, "Human-Agent Teamwork and Adjustable Autonomy in Practice," in *Proc. 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, Nara, Japan, 2003.

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

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

[18] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby, "Expressing checkable properties of dynamic systems: the Bandera Specification Language," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 1, pp. 34–56, 2002.

[19] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. New York: Springer-Verlag, 1992.

[20] E. A. Emerson, "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier, 1990, pp. 996–1072.

[21] D. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev, *Many-Dimensional Modal Logics: Theory and Applications*, ser. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2003, no. 148.

[22] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge, "Model Checking Rational Agents," *IEEE Intelligent Systems*, vol. 19, no. 5, pp. 46–52, September/October 2004.

[23] M. Kacprzak, A. Lomuscio, and W. Penczek, "Verification of Multiagent Systems via Unbounded Model Checking," in *Proc. 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IEEE Computer Society, 2004, pp. 638–645.

[24] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge, "Verifying Multi-Agent Programs by Model Checking," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 12, no. 2, pp. 239–256, March 2006.

[25] F. Raimondi and A. Lomuscio, "Automatic Verification of Multi-agent Systems by Model Checking via Ordered Binary Decision Diagrams," *Journal of Applied Logic*, vol. 5, no. 2, pp. 235–251, 2007.

[26] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge, "State-Space Reduction Techniques in Agent Verification," in *Proc. 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IEEE Computer Society, 2004, pp. 896–903.

[27] F. S. de Boer, K. V. Hindriks, W. van der Hoek, and J.-J. C. Meyer, "A verification framework for agent programming with declarative goals," *Journal of Applied Logic*, vol. 5, no. 2, pp. 277–302, 2007.

[28] M. Winikoff, "Implementing Commitment-Based Interactions," in *Proc. 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. New York, NY, USA: ACM, 2007, pp. 1–8.

[29] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer, "Programming Multi-Agent Systems in 3APL," in *Multi-Agent Programming: Languages, Platforms and Applications*, R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, Eds. Springer-Verlag, 2005, ch. 2, pp. 39–67.

[30] 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)*, ser. Lecture Notes in Computer Science, vol. 1038. Springer, 1996, pp. 42–55.

[31] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using* Jason, ser. Wiley Series in Agent Technology. John Wiley & Sons, 2007.

[32] A. P. Sistla, M. Vardi, and P. Wolper, "The Complementation Problem for Büchi Automata with Applications to Temporal Logic," *Theoretical Computer Science*, vol. 49, pp. 217–237, 1987.

[33] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple On-the-fly Automatic Verification of Linear Temporal Logic," in *Proc. 15th Workshop on Protocol Specification Testing and Verification*. Warsaw, Poland: Chapman & Hall, 1995, pp. 3–18.

[34] L. A. Dennis and B. Farwer, "Gwendolen: A BDI Language for Verifiable Agents," in *Logic and the Simulation of Interaction and Reasoning*, B. Löwe, Ed. Aberdeen: AISB, 2008, AISB'08 Workshop.

[35] "Java PathFinder: Model Java Interface (MJI)," `http://javapathfinder.sourceforge.net/The_Model_Java_Interface.html`.

[36] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, "The Maude 2.0 system," in *Rewriting Techniques and Applications (RTA 2003)*, ser. Lecture Notes in Computer Science, R. Nieuwenhuis, Ed., no. 2706. Springer-Verlag, June 2003, pp. 76–87.