Using Agent JPF to Build Models for Other Model Checkers

Louise A. Dennis, Michael Fisher, and Matt Webster

Department of Computer Science, University of Liverpool, UK
Contact: `L.A.Dennis@liverpool.ac.uk`

Abstract. We describe an extension to the AJPF agent program model-checker so that it may be used to generate models for input into other, non-agent, model-checkers. We motivate this adaptation, arguing that it improves the efficiency of the model-checking process and provides access to richer property specification languages.
We illustrate the approach by describing the export of AJPF program models to Spin and Prism. In the case of Spin we also investigate, experimentally, the effect the process has on the overall efficiency of model-checking.

1 Introduction

Agent Java Pathfinder (AJPF) [1] is a program model-checker for programs written in a range of Belief–Desire–Intention (BDI) agent programming languages. It is built on top of Java Pathfinder (JPF), an explicit state program model-checker for Java programs [2], and checks the execution of Java based interpreters for BDI languages. AJPF has a property specification language based upon Linear Temporal Logic (LTL) extended with descriptions of beliefs, intentions, etc.

AJPF (and JPF) are "program" model-checkers, meaning that they work directly on the program code, rather than on a mathematical model of the program's execution (as is typical for model-checking). In fact, these program model-checkers utilise *symbolic execution* to internally build a model to be analysed. Thus, using a *program* model-checker gives the advantage that results derived apply directly to the program under consideration. However, AJPF is slow when compared to traditional model-checkers and, in general, it is the internal generation of the program model (created by executing all possible paths through the Java program) that causes a significant bottleneck.

Hunter et al. [3] suggested the use of JPF to generate models of agent programs that could then be checked in other model-checkers. We expand upon this idea showing how AJPF can be adapted to output models in the input languages of both Spin and Prism tools. While such model generation remains slow, there are still efficiency gains, especially when the property becomes more complex. More importantly, such translations give access to a wider range of property specification languages. This means that AJPF can be used as an automated link between programs written in BDI languages and a range of model-checkers appropriate for verifying properties of those programs.

The key advantages of this approach are potential improvements in the efficiency and scope of model checking; and access to a richer set of logics for specifying program properties.

2 Background

2.1 AJPF

Java PathFinder (JPF) is an explicit state program model-checker for Java programs [2]. This means that it takes as input an executable Java program rather than a model of a Java program. It then explores all possible execution paths through this program to ensure that some property holds. For example, using JPF, it is possible to explore all possible thread scheduling options for a multi-threaded program to ensure that deadlock between threads never occurs.

AJPF [1] is a program model-checker for linear temporal logic (LTL) built on top of JPF. AJPF is specially designed for model-checking programs for rational agents, that is agents that use the BDI paradigm (see [4]) and whose execution can be described in terms of rational, goal-directed behaviour.

AJPF extends JPF with an LTL model-checking algorithm based on [7, 6][1]. The property specification language contains shallow modalities for agent concepts such as belief ($\mathcal{B}$), goal ($\mathcal{G}$), intention ($\mathcal{I}$), etc., as well as the standard LTL modalities ($\Diamond$ (eventually), $\Box$ (always), etc., but not $\bigcirc$ (next)[2]). The agent concepts are mapped to specific data structures in the Java program, and allow properties such as the following to be verified:

$$\Box\Diamond\mathcal{B}_{\mathrm{a}}\,\mathrm{reached}(\mathrm{destination})$$

This property states that *it is always the case that, eventually, agent* a *believes it has reached its destination.*

AJPF is intended for use with BDI agent programming languages which have an explicit operational semantics. The language's operational semantics is implemented in the *Agent Infrastructure Layer* (AIL): a set of Java classes that support AJPF allowing the rapid construction of interpreters for BDI agent programming languages [1]. The AIL also provides support for the Belief, Goal and Intention modalities used by the property specification language. The property specification language is discussed more fully in [1] and summarised in Fig. 1.

There are two key (and related) advantages to using a program model-checker such as AJPF instead of one with a specialised modelling language for input. Firstly, it avoids the need for the programmer (or designer) to create a separate

[1] JPF does not currently support LTL model-checking, focusing instead on searching for deadlocks and exception freedom. However it has had LTL support in the past and work is currently in progress to re-instate this support.

[2] $\bigcirc$ was omitted partly because it isn't straightforward to determine the correct semantics for "next step" in a BDI program execution and partly because it complicates the model checking algorithm.

AJPF *Property Specification Language Syntax* The syntax for property formulæ ϕ is as follows, where ag is an "agent constant" referring to a specific agent in the system, and f is a ground first-order atomic formula:

$$\phi ::= \mathcal{B}_{ag}\, f \mid \mathcal{G}_{ag} f \mid \mathcal{A}_{ag} f \mid \mathcal{I}_{ag} f \mid \mathcal{P}(f) \mid \phi \vee \phi \mid \neg\phi \mid \phi\, \mathcal{U} \phi \mid \phi \mathcal{R} \phi$$

Here, $\mathcal{B}_{ag}\, f$ is true if ag believes f to be true, $\mathcal{G}_{ag} f$ is true if ag has a goal to make f true, and so on (with $\mathcal{A}$ representing actions, $\mathcal{I}$ representing intentions, and $\mathcal{P}$ representing percepts, i.e., properties that are true in the environment).

AJPF *Property Specification Language Semantics* We summarise those aspects of the semantics of property formulæ relevant to this paper. 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. MAS is a tuple consisting of the local states of the individual agents and of the environment. Let $ag \in MAS$ be the state of an agent in the MAS tuple at this point in the program execution. Then

$$MAS \models_{MC} \mathcal{B}_{ag}\, f \quad \text{iff} \quad ag \models \mathcal{B}_{ag}\, f$$

where $\models$ is logical consequence as implemented by the agent programming language. The semantics of $\mathcal{G}_{ag} f$ and $\mathcal{I}_{ag} f$ similarly refer to internal implementations of the language interpreter. The interpretation of $\mathcal{A}_{ag} f$ is:

$$MAS \models_{MC} \mathcal{A}_{ag} f$$

if, and only if, the last action changing the environment was action f taken by agent ag. Finally, the interpretation of $\mathcal{P}(f)$ is given as:

$$MAS \models_{MC} \mathcal{P}(f)$$

if, and only if, f is a percept that holds true in the environment.

The other operators in the AJPF property specification language have standard PLTL semantics [5] and are implemented as Büchi Automata as described in [6, 7]. Thus, the classical logic operators are defined by:

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

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

$$\begin{array}{rcl} MAS \models_{MC} \quad \varphi\, \mathcal{U} \psi & \text{iff} & \text{in all runs of } P \text{ there exists a state } MAS_j \\ & & \text{such that } MAS_i \models_{MC} \varphi \text{ for all } 0 \leq i < j \\ & & \text{and } MAS_j \models_{MC} \psi \\ MAS \models_{MC} \varphi \mathcal{R} \psi & \text{iff} & \text{either } MAS_i \models_{MC} \varphi \text{ for all } i \text{ or there} \\ & & \text{exists } MAS_j \text{ such that } MAS_i \models_{MC} \varphi \\ & & \text{for all } 0 \leq i \leq j \text{ and } MAS_j \models_{MC} \varphi \wedge \psi \end{array}$$

The common temporal operators $\lozenge$ (eventually) and $\square$ (always) are, in turn, derivable from $\mathcal{U}$ and $\mathcal{R}$ in the usual way [5].

Fig. 1. Overview of the the AJPF Property Specification Language (Syntax and Semantics)

model of the implementation for verification. Secondly, in cases where certification of the program is required (e.g., [8, 9]), it increases the value of the evidence submitted to the certification authority since it provides direct information about the system that will be deployed, rather than some idealised model.

These advantages come at a cost, however. The main disadvantage of program model-checking, particularly in AJPF, is that it is very slow in comparison with existing specialised model-checkers such as Spin [10]. This has been (and continues to be) mitigated through updates to AJPF which have decreased the amount of time taken for model-checking. However, the fact remains that programs tend to be more complex than models of programs and this causes program model-checking to be much slower. Typically, to verify a program using AJPF requires minutes, hours or even days in extreme cases.

AIL-based implementations of well-known agent programming languages (e.g., GOAL [11]) are separate from the interpreters generally associated with those languages. Since, in theory, both interpreters use the same operational semantics, choosing an AIL based interpreter instead of the standard interpreter should be similar to choosing between different C compilers and an AIL interpreter can be preferred where certification is an issue. In practice, the standard interpreters are often more efficient, user-friendly and up-to-date.

One issue to consider is whether it is preferable to use just JPF to verify agent programs given that most standard interpreters are written in Java. This approach is certainly feasible, although the interpreters would probably need significant modification to work with JPF. For example, adaptations would be needed to access the AJPF Property Specification Language (or create something similar). Also, in order to minimize the state space explored by JPF careful use of Java data structures is necessary (e.g., all sets must be stored in a canonical form for state matching).

2.2 Spin

Spin [10] is a popular model-checking tool originally developed by Bell Laboratories in the 1980s. It has been in continuous development for over thirty years and is widely used in both industry and academia (e.g., [12–14]). Spin uses an input language called Promela. Typically a model of a program and the property (as a “never claim” – a sequence of transitions that should never occur) are provided in Promela, but Spin also provides tools to convert formulae written in LTL into never claims for use with the model-checker. Spin works by automatically generating programs written in C which carry out the exploration of the model relative to an LTL property. Spin’s use of compiled C code makes it very quick in terms of execution time, and this is further enhanced through other techniques like partial order reduction.

2.3 Prism

Prism [15] is a probabilistic symbolic model-checker developed primarily at the Universities of Birmingham and Oxford since 1999. Prism provides broadly similar functionality to Spin but also allows for the model-checking of probabilistic models, i.e., models whose behaviour can vary depending on probabilities built into the model. Developers can use Prism to create a probabilistic model (written in the Prism language) which can then be model-checked using Prism's own probabilistic property specification language, which subsumes several well-known probabilistic logics including PCTL, probabilistic LTL, CTL, and PCTL*. Prism has been used to formally verify a variety of systems in which reliability and randomness play a role, including communication protocols, cryptographic protocols and biological systems [16].

2.4 Related Work

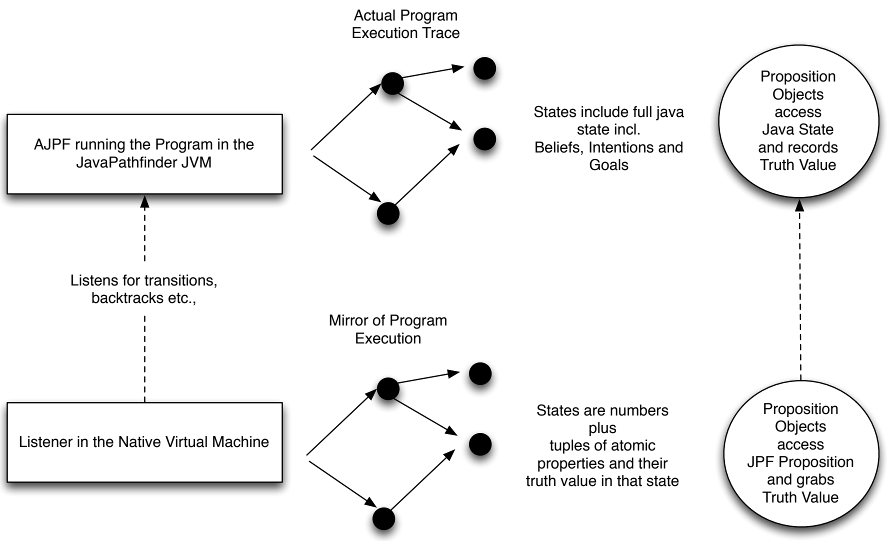
Hunter et al. [3] first suggested using JPF to generate models of programs that could then be used with alternative model-checkers. Their work targets the Brahms [17] agent programming language. They implemented a simulator for Brahms in Java and used JPF to produce a Promela model of a Brahms program. They used this system to investigate examples in air traffic control and health-care and demonstrated that it is feasible to use JPF as a model building tool. Their work did not, however, directly address the key BDI concepts of beliefs, intentions, etc., and it was a customised tool specifically aimed at the verification of Brahms programs.

The work reported here takes the ideas from Hunter et al. [3] as a starting point and aims to use them within AJPF's more generic framework in order to provide a general tool in which BDI programs, and BDI concepts can be verified in a wide range of model-checkers.

3 Exporting Models from AJPF

JPF is implemented as a specialised Java virtual machine which stores, among other things, backtracking points which allow the program model-checking algorithm to explore the entire execution space of a Java program. It is highly customisable providing numerous hooks for *listeners* that monitor and control the progress of model-checking. In what follows we will refer to the specialised Java virtual machine used by JPF as the *JPFJVM*. JPF is implemented in Java itself, therefore the JPFJVM is a program that executes in some underlying native Java virtual machine. We refer to this native virtual machine as *NatJVM*. Listeners execute in the NatJVM.

AJPF's checking process is constructed using a JPF listener. As JPF executes, it labels each state explored by the JPFJVM with a number. The AJPF listener tracks these numbers as well as the transitions between them and uses this information to construct a Kripke structure in the NatJVM. The LTL model-checking algorithm is then executed on this Kripke structure. This is partly for reasons of efficiency (the NatJVM naturally executes much faster than the JPFJVM) and also to account for the need for LTL to explore states in the



Actual Program Execution Trace
AJPF running the Program in the JavaPathfinder JVM
States include full java state incl. Beliefs, Intentions and Goals
Proposition Objects access Java State and records Truth Value
Listens for transitions, backtracks etc.,
Mirror of Program Execution
Listener in the Native Virtual Machine
States are numbers plus tuples of atomic properties and their truth value in that state
Proposition Objects access JPF Proposition and grabs Truth Value

Fig. 2. The operation of AJPF wrt. the two Java Virtual Machines

model several times if the model contains a looping path and an *until expression* (e.g., $\texttt{true}\ \mathcal{U}\ p$) exists in the LTL[3] property (see [7] and [6] for details).

In order to determine whether the agents in the original program have particular beliefs, goals, etc., it is necessary for the LTL model-checking algorithm to have access to these. However, they are not stored in the state graph of the Kripke structure accessible to the NatJVM. At the start of a model-checking run AJPF analyses the property being verified in order to produce a list of propositions that are needed for checking that property (e.g., *agent 1 believes it has reached its destination*, *agent 2 intends to win the auction* etc.). AJPF creates objects representing each of these propositions in both the JPFJVM and NatJVM. In the JPFJVM these propositional objects can access the state of the multi-agent system and explicitly check that the relevant propositions hold (e.g., that the Java object representing agent 1 contains, in its belief set, an object representing the formula *reached(destination)*).

Every time the interpreter for the agent programming language executes one step[4], all of the proposition objects are updated with their current truth value. In the NatJVM, propositional objects are created that track those in the JPFJVM. It is moderately straightforward to access an object in the JPFJVM from the

[3] "$a\mathcal{U}p$" means that "a is true continuously up until b becomes true".

[4] The meaning of a "step" in the semantics — as in the next point of interest to verification — is determined by the person implementing the semantics. Typically this is either the application of a single rule from the semantics, or of a whole reasoning cycle. This issue is discussed further in [1].

NatJVM[5]. Once an object has been accessed, inspecting the values of its fields is similarly straightforward providing they contain values of a primitive data type (such as `bool` or `int`). All this is done using JPF's Model Java Interface (MJI) interface [18] (the precise details of this implementation are specific to JPF and MJI). The implementation itself is available via the SourceForge distribution for AJPF (`http://mcapl.sourceforge.net`). The process allows, however, the modal agent properties (i.e., those related to beliefs, desires, intentions, etc.) that can be determined in the JPFJVM to be converted into state labels in the Kripke structure stored in the NatJVM. When the listener detects that a new state has been generated in the JPFJVM, the state in the Kripke structure in the NatJVM is annotated with the truth value of all the required propositions.

The process of adapting this system to produce a model for use with an alternative model checker now involves: (i) bypassing the LTL model-checking algorithm[6] but continuing to generate and maintain a set of propositional objects in order to label states in the Kripke structure, and (ii) exporting the Kripke structure in a format that can be used by another model checker.

3.1 Advantages

Ideally, a program is only model-checked once against a full set of requirements consisting of a conjunction of many properties. However, it is our experience that it is more common to check programs several times against smaller properties. For AJPF, this results in the program model being generated from the Java bytecode for each property. Our experiences with AJPF suggested that the most computationally complex part of the model-checking was in the generation of this program model, and that this was the chief cause of the slow performance of AJPF compared with other model-checkers. (This is unsurprising since in AJPF the generation of a transition in the program model can involve the symbolic execution of significant amounts of Java bytecode.)

The first advantage of the approach described above, therefore, is that exporting the program model prior to model-checking allows us to generate the program model only once, and thereafter we can use the far more compact Kripke structure representation, meaning that the time to model-check each property is reduced (on average).

The second advantage is that other model-checkers (such as Spin) have many years of development invested in an accurate and efficient implementation of LTL model-checking. Compared to model-checkers like Spin, there is a much weaker level of assurance that the LTL model-checking implemented in AJPF is correct (although it has been tested against well-known "gotchas"). Also, the

[5] The documentation for this mechanism is somewhat opaque and the process itself is complicated, but conceptually is it a simple matter of identifying the current object in the JPFJVM stack in order to obtain a reference for it. This can then be stored for future use in the NatJVM.

[6] This is not strictly necessary but it increases the speed of model generation, and avoids the pruning of some model states based on the property under consideration.

AJPF LTL model-checking algorithm is not highly optimised, being a direct adaptation of the algorithms in [7, 6]. Consequently, it seems desirable, both for reasons of confidence and efficiency, to use a more well-developed implementation of model-checking (such as Spin) where possible.

The third advantage is that this technique will allow us to use richer specification languages than LTL. For instance when verifying hybrid systems, probabilistic values frequently appear both in terms of the reliability of sensors, and the chances that an action will achieve the expected outcome. Exporting an AJPF program model into a probabilistic model-checker such as Prism will allow us to verify properties stated in more expressive logics, such as probabilistic computation tree logic (PCTL).

3.2 Disadvantages

While there are advantages to using AJPF just for model generation, there are some disadvantages as well.

Firstly, it is arguable that the direct link between the implemented program and the system being verified described in Section 2.1 has been lost. However, the LTL model-checking algorithm used in AJPF was already operating upon an automatically-generated abstraction of the system stored in the NatJVM. Therefore taking this abstracted model and exporting it to a different system does not, in our view, have a significant effect on the correctness of any verification result. However it has introduced a further step into the process which could cause an issue with software certification concerning *tool qualification*. Specifically, we have introduced another tool (Spin) to the existing verification system (AJPF) which would mean that both tools would now need to be qualified separately, and possibly again as a combined tool, with additional associated costs (tool qualification can be very costly in terms of time and finance). We do, nevertheless, provide a fully automatic route from implemented code, through an abstraction of that code, to a formal verification result, which itself is preferable to systems in which the abstraction from the implementation must be done "by hand."

Secondly, the opportunity to exploit features of the property under test in order to prune model-checking has been lost. In particular, when checking liveness properties (of the form "eventually p will happen", or $\lozenge p$) it is possible to prune the LTL model-checking search tree as soon as p occurs. It would obviously still be possible to do this, if the user were confident that only this property will be checked on the resulting model. Where the model may be used to check a number of properties such pruning is no longer a possibility and the entire program state space must be explored. Similarly, although we have not explored techniques such as property-based slicing [19] in AJPF these would also be difficult to exploit if a full model were to be exported. However, it is likely that in many cases where there are more than a few properties to be checked the additional time taken to produce a complete model will be offset by the time saved in not having to reproduce this model each time a new property needs to be verified. Similarly, the fact that we export the model as a Kripke Structure, means that we may not be able to exploit potential optimisations available within the target model checker. It should be noted, however, that some optimisations such as partial order reduction will already have been applied by JPF.

4 Exporting AJPF Models to Promela/Spin

In this section we describe the process used to translate AJPF models to Promela for verification in the Spin model-checker, and some results of Spin verification of the Promela models generated.

4.1 Translation Details

Both Spin and AJPF's LTL algorithm operate on Kripke structures so translating between the two is straightforward.

As mentioned above, within AJPF's NatJVM each state is assigned a number, e.g, `12`. This is converted to `state12` in the Spin input file. Then the list of propositional objects is examined recursively. Each proposition is converted into a simple string (without spaces or brackets), and assigned either the value true or false, depending upon its value in the state. The transitions in the AJPF model graph are kept separately from the states while Promela represents them as `goto` statements attached to states.

Example Fig. 3 shows the NatJVM model of a very simple agent program with one property (agent 1 believes "`bad`") compared to the result of exporting this model in Promela.

```
Model States:
=============

0:                                   bool bag1bad
B(ag1,bad()) = false;
                                     active proctype JPFModel()
1:                                   {
B(ag1,bad()) = false;                state0:
                                             bag1bad = false;
2:                                           goto state1;
B(ag1,bad()) = false;                state1:
                                             bag1bad = false;
Model Edges:                                 goto state2;
=============                        state2:
                                             bag1bad = false;
0-->1                                        printf("end state\n");
1-->2                                }
```

Fig. 3. Equivalent program models in AJPF (left) and Promela (right)

4.2 Results

We tested our Spin implementation on the verification of a simple "leader" agent intended to coordinate a formation of satellites as described in [20]. This program was implemented in a version of the Gwendolen language [21]. We implemented a non-deterministic environment for the agent in which messages from the satellite agents randomly arrived (or not) each time the agent took an action. This caused model-checking to explore all possible combinations of messages that the leader agent could receive. The agent was designed to assign positions to four satellites and then wait for responses. Since our hypothesis was that we would see gains in performance as the LTL property to be checked became more complex we tested the system against a sequence of properties:

1. $\Box\neg\mathcal{B}_{lead}\, bad$
   (The agent never believes something bad has happened).
2. $(\Box(\mathcal{B}_{lead}\, informed(ag1) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag1))) \rightarrow \Box\neg\mathcal{B}_{lead}\, bad$
   (If it is always the case that when the leader has informed agent 1 of its position then eventually the leader will believe agent 1 is maintaining that position, then it is always the case that the leader does not believe something bad has happened).
3. $(\Box(\mathcal{B}_{lead}\, informed(ag2) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag2))) \wedge \Box(\mathcal{B}_{lead}\, informed(ag1) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag1))) \rightarrow \Box\neg\mathcal{B}_{lead}\, bad$
4. $(\Box(\mathcal{B}_{lead}\, informed(ag3) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag3)) \wedge \Box(\mathcal{B}_{lead}\, informed(ag2) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag2)) \wedge \Box(\mathcal{B}_{lead}\, informed(ag1) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag1))) \rightarrow \Box\neg\mathcal{B}_{lead}\, bad$
5. $(\Box(\mathcal{B}_{lead}\, informed(ag4) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag4)) \wedge \Box(\mathcal{B}_{lead}\, informed(ag3) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag3)) \wedge \Box(\mathcal{B}_{lead}\, informed(ag2) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag2)) \wedge \Box(\mathcal{B}_{lead}\, informed(ag1) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag1))) \rightarrow \Box\neg\mathcal{B}_{lead}\, bad$
6. $(\Box(\mathcal{B}_{lead}\, informed(ag4) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag4)) \wedge \Box(\mathcal{B}_{lead}\, informed(ag3) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag3)) \wedge \Box(\mathcal{B}_{lead}\, informed(ag2) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag2)) \wedge \Box(\mathcal{B}_{lead}\, informed(ag1) \rightarrow \Diamond\mathcal{B}_{lead}\, maintaining\_pos(ag1))) \wedge \Box(\mathcal{B}_{lead}\, formation(square) \rightarrow \Diamond\mathcal{B}_{lead}\, informed(ag1))) \rightarrow \Box\neg\mathcal{B}_{lead}\, bad$

This sequence of increasingly complex properties was constructed so that each property had the form $P_1 \wedge \ldots \wedge P_n \rightarrow Q$ for some $n \geq 0$ and each P_i was of the form $(\Box(P'_i \rightarrow \Diamond Q_i))$. With the addition of each such logical antecedent the property automata became considerably more complex. Furthermore, the antecedents were chosen so that we were confident that on at least some paths through the program P'_i would be true at some point, necessitating that the LTL checker explore the product automata for $\Diamond Q_i$. We judged that this sequence of properties provided a good test for the way each model-checker's performance scaled as the property under test became more complicated.

Spin model-checking requires a sequence of steps to be undertaken: the LTL property must be translated to a "never claim" (effectively representing the automaton corresponding to the negation of the required property), then it is compiled together with the Promela description into C, which is then compiled again before being run as a C program. We used the Ltl3ba tool [22] to compile the LTL property into a never claim since this is more efficient than the built-in Spin compiler. In our results we present the total time taken for all Spin operations (Spin Time) and the total time taken overall including generation of the model in AJPF.

Property	AJPF	Spin		
		AJPF model generation	Spin Time	Total Overall Time
1	5m25s	5m17s	1.972s	5m19s
2	5m54s	5m50s	3.180s	5m53s
3	7m9s	6m28s	4.369s	6m32s
4	8m50s	7m34s	6.443s	7m40s
5	9m22s	8m27s	10.015s	8m37s
6	—	8m51s	22.361s	9m13s

Table 1. Results Comparing AJPF with and without Spin Model Checking

Table 1 shows the running times for model-checking the six properties on a 2.8 GHz Intel Core i7 Macbook running MacOS 10.7.4 with 8 GB of memory. Fig. 4 shows the same information as a graph. There is no result for AJPF model-checking of the final property since the system suffered a stack overflow error when attempting to build the property automata.

The results show that as the LTL property becomes more complex, model-checking using the AJPF to Promela/Spin translation tool is marginally more efficient than using AJPF alone. It should be noted that in the Spin case, where AJPF is not performing LTL model-checking, and is using a simple list of propositions (rather than an LTL property) the time to generate the model still increases as the property becomes more complex. This is explained by the overhead involved in tracking the proposition objects in the JPFJVM and the NatJVM: as more propositions are involved this time increases.

If only one AJPF model were to be generated then Spin would give considerable time savings overall. (NB. In this case it would need to be the AJPF model with all relevant propositions, i.e., the one taking nearly 9 minutes to generate.)

We note that the simple fact that AJPF cannot generate a property automata for property 6 is a compelling argument that combining AJPF with Spin or some other model-checker is sometimes necessary. It also illustrates the point that Spin is well optimised for working with LTL where AJPF is not.

5 Exporting AJPF Models to Prism

5.1 Translation Details

Both AJPF's NatJVM and Spin operate on Kripke structures so it was a straightforward process to translate between them. The Prism input language

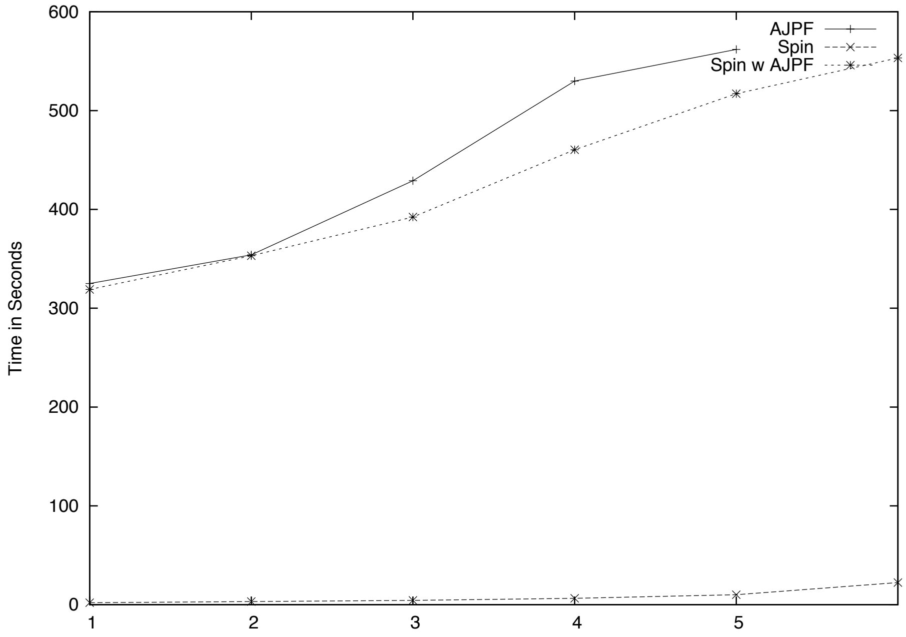


Fig. 4. Results of AJPF against Spin

is based on probabilistic timed automata. In the examples we are particularly interested in exploring, we can consider the model to be a Kripke structure enhanced by labels on the transitions representing probabilities.

We therefore needed to make some alterations to AJPF. JPF, and hence AJPF, is able to branch the search space when a random element is selected from a finite set. However the system does not record the probabilities of each branch created in a manner accessible to the NatJVM. We developed a new class `Choice` in Java which represented a probabilistic choice from a finite set of options. This class provided a method `pickChoice` which would perform a choice on a probabilistic basis. If this class was used in programming at the JPFJVM level, then a NatJVM *native peer* could detect invocations of methods in this class, intercept such invocations and use a customised *choice generator*, to branch the search space in the JPFJVM while annotating the edges of the model graph in the NatJVM with the appropriate probabilities. The use of native peers and choice generators are standard JPF customisation processes for controlling and recording search in model-checking (see [18] for a discussion of their use). In short, programming with the `Choice` class, in the normal execution of the program, simply picks an element from a set based on some probability distribution. When executed within AJPF, the `Choice` class causes the system to explore all possible choices and label each branch with its probability.

After this the process of translating these models into Prism's input language is straightforward.

1. First we initialise the model: We input it as a discrete time Markov chain (`dtmc`); We list the numbers of all states and state the initial state (`0`); We list all the properties initialise them to false.
2. We then iterate through the states in the AJPF model. For each state we:
   (a) Print out `state = num` where `num` is the state number.
   (b) Iterate over all its outgoing edges, for each edge:
      i. Print out the probability of that edge being traversed
      ii. Print out the state number, and values of the properties for the state at the far end of the edge.

As an example we consider a simple program based on [8] in which an unmanned aerial vehicle (UAV) must detect potential collisions. The UAV's radar is only 90% reliable, so it does not always perform an 'evade' maneouvre when a collision is possible. The agent controlling the UAV is implemented in Gwendolen which does not contain any probabilistic aspects. However the agent was placed in an environment programmed in Java and we used the `Choice` class to represent the unreliability of the sensor when the agent requested incoming perceptions [7].

The model is tracking two properties $\mathcal{P}(collision)$ which means *a potential collision is perceptible in the environment* and $\mathcal{A}_{uav}evade$ which means *the last action performed was the uav agent taking an evade maneouvre.* The agent was programmed to make evade maneouvres when it believed there would be a collision. It only believed there would be a collision if a potential collision was perceptible and the sensor conveyed that information to the agent.

A fragment of the AJPF model for this program, adapted to show the probability of transitions is shown in Fig. 5 alongside the full model exported to the Prism input language[8]. Fig. 6 gives a brief outline of some key features of Prism's property specification language, its full semantics can be found in [23].

5.2 Results

We do not provide performance results since AJPF and Prism are incomparable using, as they do, different input languages (AJPF does not support probabilistic reasoning and Prism does not support non-probabilistic LTL model checking). We model-checked the above program in Prism against the property

$$\mathtt{P}^{=?}\Box(\mathcal{P}(collision) \rightarrow \Diamond\mathcal{A}_{uav}evade)$$

to establish that the probability that the UAV would evade a collision, if one were possible, was 90%.

We also investigated a more complex model, again based on [8], in which the probability of a potential collision arising was also 90% (where it was certain

[7] We would also be able to investigate properties of BDI programming languages with probabilistic features, providing their AIL implementation used the `Choice` class — see Further Work.

[8] Note that the nature of rounding in Java means that 0.1 is, in several places, represented as 0.09999999999999998.

AJPF Model

```
Model States:
=============

....

3:
A(uav,evade()) = false;
P(collision()) = false;

4:
A(uav,evade()) = false;
P(collision()) = true;

5:
A(uav,evade()) = true;
P(collision()) = false;              Model Edges:
                                     =============
6:
A(uav,evade()) = true;               ...
P(collision()) = false;
                                     0.9 ::: 3-->4
7:                                   0.09999999999999998 ::: 3-->12
A(uav,evade()) = true;               1.0 ::: 4-->5
P(collision()) = true;               1.0 ::: 5-->6
                                     0.9 ::: 6-->7
...                                  0.09999999999999998 ::: 6-->10
```

Prism Model

```
dtmc

module jpfModel
state : [0 ..13] init 0;
auavevade: bool init false;
pcollision: bool init false;
[] state = 1 -> 1.0:(state'=2) & (auavevade'= false) & (pcollision'= false);
[] state = 2 -> 1.0:(state'=3) & (auavevade'= false) & (pcollision'= false);
[] state = 3 -> 0.9:(state'=4) & (auavevade'= false) & (pcollision'= true)
   + 0.09999999999999998:(state'=12) & (auavevade'= false) & (pcollision'= true);
[] state = 4 -> 1.0:(state'=5) & (auavevade'= true) & (pcollision'= false);
[] state = 5 -> 1.0:(state'=6) & (auavevade'= true) & (pcollision'= false);
[] state = 6 -> 0.9:(state'=7) & (auavevade'= true) & (pcollision'= true)
   + 0.09999999999999998:(state'=10) & (auavevade'= true) & (pcollision'= true);
[] state = 7 -> 1.0:(state'=8) & (auavevade'= false) & (pcollision'= false);
[] state = 8 -> 1.0:(state'=9) & (auavevade'= false) & (pcollision'= false);
[] state = 10 -> 1.0:(state'=11) & (auavevade'= false) & (pcollision'= false);
[] state = 11 -> 1.0:(state'=9) & (auavevade'= false) & (pcollision'= false);
[] state = 12 -> 1.0:(state'=13) & (auavevade'= false) & (pcollision'= false);
[] state = 13 -> 1.0:(state'=9) & (auavevade'= false) & (pcollision'= false);
endmodule
```

Fig. 5. Comparison of Models for AJPF and Prism

The syntax of the fragment of the PRISM property specification language relevant here is given by the following grammar:

$$\phi ::= \texttt{true} \mid \texttt{a} \mid \phi \wedge \phi \mid \neg\phi \mid \texttt{P}^{\bowtie p}[\psi]$$
$$\psi ::= \phi \texttt{U} \phi$$

where $\texttt{a}$ is an atomic proposition, $\bowtie \in \{\leq, <, \geq, >\}$, $p \in \mathbb{Q}_{\geq 0}$, and $k \in \mathbb{N}$.

The semantics of the propositional logic statements and the CTL until operator are standard and allow $\square$ (always) and $\lozenge$ (eventually) to be defined. $\texttt{P}$ is a probabilistic operator and indicates the probability that some property is true along all paths from some state s where the operator is evaluated. For instance $\texttt{P}^{\geq 0.98}\psi$ means "the probability that ψ is satisfied by the paths from state s is greater than 0.98".

It is also possible to take a quantitative approach so $\texttt{P}^{=?}\psi$ will return a value for the probability that ψ is satisfied for all paths from state s.

Fig. 6. The PRISM Property Specification language

in the simple model above) and the UAV had to interact with an air traffic control agent, and go through take off procedures. The environment contained a navigation manager which, on a probabilistic basis, would either tell the UAV to change its current heading or land. In this situation the probability of the UAV making an evade maneouvre when a collision was perceptible (rather than landing, or spontaneously changing its heading following an instruction from the navigation manager) dropped to 30%.

6 Further Work

One of our primary motivations in performing this work was to enable the probabilistic model-checking of BDI agents, particularly in practical health-care and hybrid systems scenarios. We intend therefore to explore more sophisticated and realistic examples in which an implemented BDI based agent program is executed in AJPF and then model-checked in PRISM. Our interest is in producing results about the overall reliability of systems based on probabilistic analyses of the reliability of sensors and actuators derived through testing.

We are also interested in exploring the verification of multi-agent properties involving strategies. This would involve both adapting our output format for an ATL model-checker, such as MCMAS [24], and adapting the internal models so that transitions are labelled with actions. We may also wish to extend the AIL so that agents can explicitly reason about their own strategies. We would also like to investigate the verification of properties of BDI programming languages that incorporate probabilistic features, something which will likely require that their AIL implementation uses the `Choice` class.

It would also be possible to adapt AJPF to save and then re-import its own models, avoiding the model generation bottleneck while retaining the entire verification process within a single system. While this would lose some of the benefits (e.g., assurance and efficiency), it would provide a simpler tool and might be more attractive in certification situations.

7 Conclusion

We have shown how the ideas of Hunter et. al [3] for the use of JPF to generate models of Brahms programs for export into SPIN, can be generalised and integrated within the AJPF tool for model-checking BDI programs.

This provides a generic tool for generating models of agent programs implemented in a wide range of BDI languages. These models can then be exported into the input languages of the model-checker of choice. Where such a model-checker operates on Kripke structures there is a direct translation from AJPF's own internal model to that of the target model-checker. For model-checkers using richer input structures it is still relatively easy, using the customisation options available with JPF, to enrich AJPF's models so that they can be exported appropriately. We provided an example of one such adaptation allowing BDI programs to be probabilistically model-checked via the PRISM model-checker.

Acknowledgments. Work partially funded by EPSRC through the "Trustworthy Robotic Assistants", and "Reconfigurable Autonomy" projects, and by the ERDF/NWDA-funded Virtual Engineering Centre.

References

1. Dennis, L.A., Fisher, M., Webster, M., Bordini, R.H.: Model Checking Agent Programming Languages. Automated Software Engineering **19**(1) (2012) 5–63
2. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering **10**(2) (2003) 203–232
3. Hunter, J., Raimondi, F., Rungta, N., Stocker, R.: A Synergistic and Extensible Framework for Multi-Agent System Verification. In Ito, T., Jonker, C., Gini, M., Shehory, O., eds.: Proc. 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS (2013) 869–876
4. Wooldridge, M.: Reasoning about Rational Agents. The MIT Press (2000)
5. Emerson, E.A.: Temporal and Modal Logic. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Elsevier (1990) 996–1072
6. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proc. 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, London, UK, UK, Chapman & Hall, Ltd. (1996) 3–18
7. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient Algorithms for the Verification of Temporal Properties. In: Formal Methods in System Design. (1992) 275–288
8. Webster, M., Fisher, M., Cameron, N., Jump, M.: Formal Methods and the Certification of Autonomous Unmanned Aircraft Systems. In: Proc. 30th International Conference on Computer Safety, Reliability and Security (SAFECOMP). Volume 6894 of LNCS., Springer (2011) 228–242

9. Webster, M., Cameron, N., Jump, M., Fisher, M.: Generating Certification Evidence for Autonomous Unmanned Aircraft Using Model Checking and Simulation. Journal of Aerospace Computing, Information, and Communication (2013)
10. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
11. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.J.: Agent Programming with Declarative Goals. In: Intelligent Agents VII. Volume 1986 of LNAI., Springer (2001) 228–243
12. Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W., White, J.L.: Formal Analysis of the Remote Agent Before and After Flight. In: Proc. 5th NASA Langley Formal Methods Workshop, Virginia, USA. (2000)
13. Kars, P.: The Application of Promela and Spin in the BOS Project (Abstract) (1996) http://spinroot.com/spin/Workshops/ws96/Ka.pdf. Accessed 2013-05-30.
14. Kirsch, M.T., Regenie, V.A., Aguilar, M.L., Gonzalez, O., Bay, M., Davis, M.L., Null, C.H., Scully, R.C., Kichak, R.A.: Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation. NASA Engineering and Safety Center Technical Assessment Report (January 2011)
15. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic Symbolic Model Checker. In: Proc. 12th Int. Conf. Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS). Volume 2324 of LNCS., Springer (2002)
16. PRISM: Probabilistic Symbolic Model Checker: http://www.prismmodelchecker.org/. Accessed 2013-05-31.
17. Sierhuis, M., Clancey, W.J.: Modeling and Simulating Work Practice: A Method for Work Systems Design. IEEE Intelligent Systems **17**(5) (2002) 32–41
18. JPF... the Swiss Army Knife of Java^{TM} verification: http://babelfish.arc.nasa.gov/trac/jpf/. Accessed 2013-06-09.
19. Bordini, R.H., Fisher, M., Wooldridge, M., Visser, W.: Property-based slicing for agent verification. J. Log. and Comput. **19**(6) (December 2009) 1385–1425
20. Lincoln, N.K., Veres, S.M., Dennis, L.A., Fisher, M., Lisitsa, A.: Autonomous Asteroid Exploration - Agent Based Control for Autonomous Spacecraft in Complex Environments. IEEE Computational Intelligence Magazine (To appear) Special Issue on Computational Intelligence for Space Systems and Operations.
21. Dennis, L.A., Farwer, B.: Gwendolen: A BDI Language for Verifiable Agents. In: Proc. AISB Workshop on Logic and the Simulation of Interaction and Reasoning, AISB (2008)
22. Babiak, T., Kretínský, M., Rehák, V., Strejcek, J.: LTL to Büchi Automata Translation: Fast and More Deterministic. In: Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 7214 of LNCS., Springer (2012) 95–109
23. Norman, G., Parker, D., Sproston, J.: Model checking for probabilistic timed automata. Formal Methods in System Design (2012) 1–27
24. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. In: Proc. 21st International Conference on Computer Aided Verification (CAV). (2009)