Two-Stage Agent Program Verification

Louise A. Dennis Michael Fisher 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 potentially 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 both the SPIN and PRISM model-checkers. We also investigate, experimentally, the effect the process has on the overall efficiency of model-checking.

Keywords: Model checking, BDI agent programming, AJPF, SPIN, PRISM.

1 Introduction

Agent Java Pathfinder (AJPF) [7] is a 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 [29], and exhaustively 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 standard model-checking). Using a *program* model-checker gives the advantage that results derived apply directly to the program under consideration without the need for an intermediate stage. However, such program model-checkers utilise *symbolic execution* to internally build a model to be analysed and, consequently, AJPF is slow when compared to traditional model-checkers. It is typically 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. [16] suggested alleviating this by using JPF to generate models of agent programs that could then be checked in other model-checkers. The goal of this paper is to expand upon this idea showing how AJPF can be adapted to output models in the input languages of both SPIN and PRISM tools. Model generation remains slow,

and it is unclear that efficiency improves on individual runs, though there will be gains if one model is reused several times to check different properties. More importantly, such translations give access to a wider range of property specification languages. Consequently, 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.

We are particularly interested in applying program model-checking to the verification of hybrid systems in which a BDI agent program controls a physical system consisting of sensors, actuators and control systems [9]. Such systems necessarily involve probabilistic information about sensor and actuator reliability and the end results of verification are therefore theorems involving probabilities. For instance we have been considering the verification of a robot-to-human handover task in which a robot has to pass a table leg to a person (see, e.g., [11]). When the person gets the table leg they will fix it to the table top. The end goal is for the robot and human to work together to manufacture a complete table. In order for the robot to let go of the table leg, it must be sure the person is ready to take hold of the leg, otherwise the leg could be dropped (it determines this using several factors such as the the person's gaze, and the location of their hand). Given probabilistic information about the behaviour of people and sensors involved in the task we would like to be able to formally verify (or formally discover) properties such as the following:

- What is the probability that eventually the robot will drop some object such as the table leg?
- What is the probability that eventually the leg will be fixed to the table?

The key advantages of the approach outlined in this paper 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 model-checker for Java programs [29]. It is a *program* model-checker, meaning that it takes as input an executable Java program rather than a model of a Java program and then exhaustively 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 [7] is, in turn, a program model-checker built on top of JPF. AJPF is specially designed for model-checking programs for agents that use the BDI paradigm (see [31]) and whose execution can be described in terms of rational, goal-directed behaviour. AJPF extends JPF with a linear temporal logic (LTL) model-checking algorithm based on [4, 10][1]. Crucially, the property specification language contains

[1] JPF does not currently support LTL model-checking, focusing instead on searching for deadlocks and exception freedom. Work is currently in progress to re-integrate this support.

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) and $\Box$ (always)[2]. The BDI agent concepts [26] are mapped to specific data structures in the Java program, allowing 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. This operational semantics is implemented in the *Agent Infrastructure Layer* (AIL), a set of Java classes supporting AJPF and allowing the rapid construction of interpreters for BDI agent programming languages [7]. The AIL also provides support for the Belief, Goal and Intention modalities used by the formal property specification language. This language is discussed more fully in [7] and summarised in Appendix A. Note, crucially, that temporal operators cannot be nested within the belief, goal and intention operators.

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, this approach avoids the need for the programmer (or designer) to create a separate model of the implementation for verification purposes. Secondly, in cases where certification of the program is required (e.g., [30]), this approach 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. 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 [15]. This has been (and continues to be) mitigated through updates to AJPF which have decreased the 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 slower[3]. Typically, to verify a program using AJPF requires minutes, hours or even days in extreme cases.

AIL provides a framework for implementing a wide range of well-known agent programming languages (e.g., GOAL [14]). Typically, agent programming languages are separate from the interpreters generally associated with those languages. Since different interpreters will use the same operational semantics, choosing an AIL-based interpreter instead of the standard interpreter should be similar to choosing between different C compilers. An AIL interpreter can be preferred, therefore, 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 likely need significant modification

[2]The next operator '$\bigcirc$' was omitted partly because it is not always simple to determine the correct semantics for "next step" in a BDI program execution and partly because its omission simplifies the model checking algorithm.

[3]This is to be expected, since AJPF combines *explicit-state* representation with the use of *symbolic execution* to explore the possible behaviours.

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 [15] 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., [13, 18, 19]). SPIN uses an input language called PROMELA. Typically a model of a program and the property (as a "never claim" — an automaton describing executions that violate the property) are both 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 such as partial order reduction. In this paper we use SPIN version 6.2.3 (24 October 2012).

2.3 PRISM

PRISM [20] is a probabilistic symbolic model-checker in continuous development since 1999, primarily at the Universities of Birmingham and Oxford. 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 represented in 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 uncertainty play a role, including communication protocols, cryptographic protocols and biological systems [25]. In this paper we use PRISM version `4.1.beta2`.

2.4 Related Work

As mentioned in the introduction, Hunter et al. [16] first suggested using JPF to generate models of programs that could then be used with alternative model-checkers. Their work targets the Brahms [27] 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 healthcare 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. Their tool also contains support for the export of models to PRISM and NuSMV. In theory the framework can be applied to any multi-agent system, not just those imple-

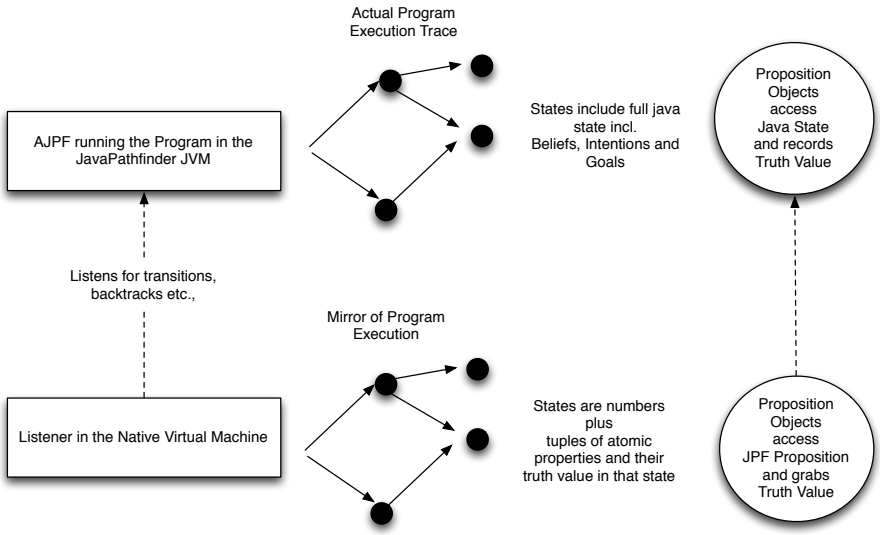

Figure 1: The operation of AJPF wrt. the two Java Virtual Machines

mented in Brahms, though no explicit support exists for adapting systems for such use in a generic way.

The work here takes the ideas from Hunter et al. [16] as a starting point and aims to use them within AJPF's more generic framework in order to provide a general open source tool in which BDI programs can be verified in a range of model-checkers and which allows BDI concepts such as beliefs and goals to be easily and explicitly referred to as part of the specification of properties in a range of input languages.

This work is an extension of a previous workshop paper by the same authors [6]. In this paper we provide more details of the implementation. In particular this paper describes further work with the PRISM model checker which adds a new case study and discusses the time and memory resources used during verification.

3 Generating Program Models Using AJPF

JPF is implemented via a specialised Java virtual machine which stores, among other things, backtracking points. This allows the program model-checking algorithm to explore the entire execution space of a Java program. It is highly customisable, providing numerous hooks for Java *Listener*s 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 unique 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 model several times if the model contains a looping path and an *until expression* (e.g., `true` $\mathcal{U}$ p) exists in the LTL[4] property (see [4] and [10] for details).

In order to determine whether the agents have particular beliefs, goals, etc., it is necessary for the LTL model-checking algorithm to have access to these. However, these structures exist in the JPFJVM not the NatJVM and so techniques (described in detail below) are required to create objects that represent propositions of interest (e.g., "agent 1 believes the formation is a square") in the JPFJVM, and then track these from the NatJVM in order to label the states in the Kripke structure appropriately.

The process of adapting this system to produce a model for use with an alternative model checker involves: (i) bypassing the LTL model-checking algorithm within AJPF[5] 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 subsequently be used by another model checker.

At the start of a model-checking run AJPF analyses the property being verified in order to produce a list of logical 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 then 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)*).

In detail, the system maintains three different types of objects representing non-temporal propositions, one in the NatJVM (*native propositions*) and two in the JPFJVM (*abstract* and *concrete propositions*). It is not strictly necessary to maintain two in the JPFJVM but the details of how the three different types of proposition are created during parsing means that abstract propositions are created first (in both JVMs) and linked by storing a reference to the JPFJVM version in the NatJVM. Once that is done, native propositions are created from the abstract propositions in the NatJVM while concrete propositions are created from them in the JPFJVM.

When the NatJVM accesses an object in the JPFJVM using a reference (as the native propositions access their corresponding abstract propositions), inspecting the values of its fields is straightforward providing they contain values of a primitive data type (such as `bool` or `int`). This is achieved using JPF's Model Java Interface (MJI) interface [17]. The implementation is available via AJPF's SourceForge distribution[6].

In the JPFJVM the concrete propositions have methods for checking their truth against the current agent system. These concrete propositions update a Boolean field

[4] "$a\mathcal{U}p$" means that "a is true continuously until b becomes true"; see Appendix D.

[5] 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.

[6] `http://mcapl.sourceforge.net`

in their corresponding abstract proposition whenever their own truth is checked.

In the NatJVM a Büchi Automaton is constructed from the property. This is the finite-state automaton that will be used for checking the truth of the property during model-checking. When checking the truth value of an individual state in the Büchi Automaton, at a particular point in an execution, only the truth value of propositions are checked. Evaluating the truth of temporal properties associated with the state is deferred for further exploration of the automaton. Therefore each Büchi state maintains a list of native proposition objects, and, when the truth of the state is checked these consult the fields of their corresponding object in the JPFJVM.

Each time the interpreter for the agent programming language executes one step[7], all of the concrete proposition objects check their truth and update the truth value field in the abstract propositions. Precisely when this occurs is the choice of the interpreter designer. It is typically either each time a transition is made in the operational semantics, or each time a full reasoning cycle in the operational semantics completes.

Properties in the NatJVM are updated whenever JPF determines that a transition has been made in the program running in the JPFJVM. When used in conjunction with partial order reduction JPF typically detects a transition when there is a scheduling choice between agents (and possibly the environment) or branching caused by the invocation of some random choice. It is at this point, therefore, that the Native-level proposition objects examine the relevant fields in the abstract objects stored in the JPFJVM and update their own fields. This process is illustrated informally in Figure 2.

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 multiple times, once 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 here, 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 these, 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 pitfalls). Also, the AJPF LTL model-checking algorithm is not

[7]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 [7].

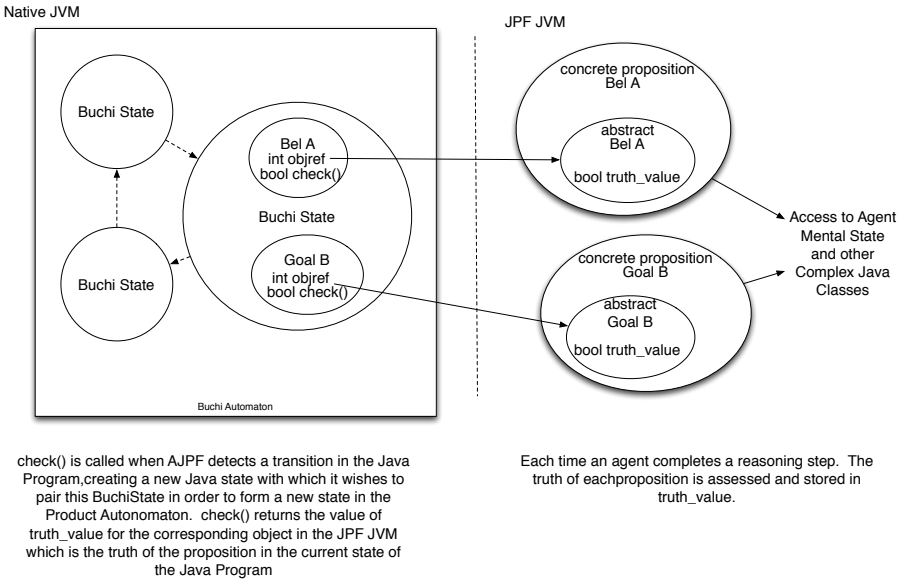

Figure 2: The relationship between proposition classes in AJPF

highly optimised, being a direct adaptation of the algorithms in [4, 10]. Consequently, it is desirable, 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 chance that an external action will achieve its 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 clearly 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. Taking this abstracted model and exporting it to a different system does not, in our view, have a significant effect on the overall 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 exist-

ing 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 $\Diamond p$) 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 yet explored techniques such as property-based slicing [3] 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 several 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 well-known optimisation techniques, such as partial order reduction, are implemented in JPF and so are applied during the model generation phase, hence the Kripke structure is already in an optimised form.

4 Exporting AJPF Models to SPIN

In this section we describe the detailed 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 similar automaton structures so translating between the two is straightforward. In AJPF a model can be viewed as a set of model states, ms, which are a tuple of an integer, i, and a set of propositions, P. The model itself includes a function, F, that maps an integer (representing a particular model state) to a set of integers (representing all the model states which can be reached in one transition). In this way the model describes a graph.

Since, 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. PROMELA represents the transitions between states as `goto` statements attached to states.

The process of translating these models into PROMELA is straightforward:

States:

```
0: B(ag1,bad()) = false;

1: B(ag1,bad()) = false;

2: B(ag1,bad()) = false;
```

Edges:

```
0-->1
1-->2
```

```
bool bag1bad;

active proctype JPFModel() {
  state0: bag1bad = false;
          goto state1;
  state1: bag1bad = false;
          goto state2;
  state2:bag1bad = false;
          printf("end state\n");
}
```

Figure 3: Equivalent program models in AJPF (left) and PROMELA (right)

1. First we initialise the model: we convert all the properties in the model to strings (as described above) and print these as a list of boolean variables ("`bool`").
2. Print `active proctype JPFModel() {`.
3. We then iterate through the states in the AJPF model. For each state we carry out the following:
   (a) Print out `state`*num*`:` where *num* is the state number.
   (b) Iterate over all the propositions printing $prop_s$ `= true` or $prop_s$ `= false` as appropriate (where $prop_s$ is the string representing the proposition).
   (c) If there is more than one edge print `if`.
   (d) Iterate over all the state's outgoing edges, print `goto state`*num*`;` where *num* is the number of the next state.
   (e) If there is more than one edge print `fi;`
   (f) If there are no outgoing edges print `printf("end state\n")`.
4. Print }.

Figure 3 shows the NatJVM model of a simple agent program with one property (agent 1 believes the proposition "`bad`") compared to the result of exporting this model in PROMELA.

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 [22]. This program was implemented in a version of the GWENDOLEN BDI language [5]. We implemented a non-deterministic environment for the agent in which messages from the satellite agents could randomly arrive 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 lead 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).

The next three properties increase in complexity by adding subformulae for agents $ag2$, $ag3$ and $ag4$. The final property adds another subformula which says that it is always the case that if the leader believes that the formation is in the shape of a square, then eventually it believes that it has informed agent $ag1$ of this.

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 [1] 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		AJPF + SPIN			
	Time	Memory	AJPF Time	Memory	SPIN Time	Total Time
1	11m20s	413MB	11m17s	413MB	4s	11m21s
2	13m09s	410MB	13m04s	410MB	5s	13m09s
3	15m27s	410MB	15m30s	410MB	7s	15m37s
4	18m19s	408MB	18m18s	411MB	11s	18m29s
5	22m14s	411MB	21m54s	417MB	16s	22m10s
6	—	—	22m3s	406MB	24s	22m27s

Table 1: 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. 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 less 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. In fact it is clear that the number of propositions are the major factor affecting the efficiency of the model checking – not the complexity of the temporal expressions within the property itself. Given that the SPIN version has additional overheads (the model needs to be written to a file and then SPIN itself needs to be run) the overall time taken to model check tends to be slower, even if the time taken to build the model is faster. If, however, a model is to be generated once and then checked against a number of properties then using SPIN together with AJPF is clearly preferable.

It is interesting to note that AJPF could not generate a property automaton for property 6. Indeed, this 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 optimised for working with LTL where AJPF is not.

5 Exporting AJPF Models to PRISM

This section describes the translation of AJPF models to PRISM's input language.

5.1 Translation Details

Both AJPF's NatJVM and SPIN operate on Kripke structures so it was a straightforward process to translate between them. However, the PRISM input language is based on probabilistic timed automata, structures that are commonly used to model systems that exhibit both timed and probabilistic behaviour, such as network protocols, sensors,

```
public class Choice<O extends Object> {
        public ArrayList<Option<O>> choicelist = new ArrayList<Option<O>>();
        public double thischoice_probability;
        public Random r = new Random();

        // Pick a choice according to a probability distribution
        public O choose() {
                int i = pickChoice(choicelist.size() - 1);
                thischoice = choicelist.get(i).getProb();
                O choice = choicelist.get(i).getObj();
                return choice;
        }

        // pickChoice performs roulette wheel selection over the choices
        // it returns an int and takes an int as a parameter in order to
        // simplify interactions with the Native JVM.
        public int pickChoice(int limit) {
                double rvalue = r.nextDouble();
                int list_index = 0;
                Option<O> current = choicelist.get(list_index);
                double accumulator = current.getProb();
                while (rvalue > accumulator) {
                        list_index++;
                        if (list_index == limit) { break; }
                        current = choicelist.get(list_index);
                        accumulator += current.getProb();
                }
                return list_index;
        }

        // The Option Class representing a tuple of a probability and
        // the value associated with it.
        public class Option<O1 extends Object> {
                public double probability;
                public O1 value;
                public double getProb() { return probability; }
                public O1 getObj() { return value; }
        }
}
```

Figure 4: The Choice Class (Simplified)

biological models, etc. While we do not utilise the *timing* dimension here, the *probabilistic* aspect is important. The key difference between the automata considered earlier and their probabilistic counterparts is that transitions between states are now probabilistic. Specifically, such automata typically incorporate a probability distribution to inform the choice amongst the potential transitions [21]. Consequently, information about this probability distribution is important in constructing probabilistic automata.

In order to support transitions with probability labels, it was necessary 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.

In order to address this we made use of a JPF customisation tool known as a *native peer*. The native peer of a Java object can intercept the execution of particular methods associated with the object. When a method is intercepted, alternative code associated with the native peer is executed in the NatJVM instead of the existing code associated with the object. This can allow complex algorithms to be executed natively for efficiency reasons or, as is the case here, to control branching in the program model.

We developed a new class, `Choice`, in Java which represented a probabilistic choice from a finite set of options. We also developed a native peer for this class.

A `Choice` object consists of an array of `Options`. An `Option` is a tuple comprising both a probability and a value (of whatever class is needed for the results of the choice). The probabilities of the options in the array add up to one (at least in theory). At a high level, when asked to pick a choice the class returns one of the options from the array. When not executing in JPF, the class selects the option by using a standard "roulette wheel" algorithm to select an option according to the probability distribution. When executing in JPF, the method that performs roulette wheel selection is intercepted and, instead, a *choice generator* is created. This sets a backtrack point in the system and each time the execution returns to that backtrack point a different option is selected until all choices have been explored. The `Choice` class maintains, as a field, the probability of the current choice allowing this to be accessed by the AJPF Listener and used to annotate the edges of the model.

Figure 4 shows a simplified version of the Java code for the `Choice` and `Option` classes. When asked to pick a choice, the class calls first its `choose` method, which in turn calls the `pickChoice` method. `pickChoice` returns an index to the `Option` array. `choose` then selects the relevant option from the array and returns it to the rest of the program. We used a two stage process because it allowed us to deal just with primitive datatypes in the `pickChoice` method (which made programming the native peer considerably simpler). When not executing in JPF, `pickChoice` uses a roulette wheel algorithm to select an option. When the `choose` method is invoked outside AJPF, therefore, the effect is to randomly return one of the values from the list according to the distribution specified. Once `pickChoice` has returned a value, then `choose` updates the field, `thischoice_probability`, with the current probability and returns the relevant option to the program.

We cannot use the generation of a random double-precision floating point number to branch the search space in JPF since there are 2^{64} choices and the search space would increase in size considerably. Instead, we branch the search space with one branch for each of the possible options in the `Choice` class. This is done by using a native peer

for the `Choice` class a (very simplified) version of which is shown in Figure 5. When running in JPF, the native peer intercepts calls to `pickChoice` and creates a choice generator (a branch point in the program automaton) with one branch for each index to the `Option` array. The version of `pickChoice` in the JPFJVM is not executed and instead the version in the native peer is used. Each branch of the choice generator returns a different index to the `Option` array. In this way the exploration of successive branches causes every index to be returned to the `choose` method.

```
public class JPF_ajpf_util_Choice<O extends Object> extends NativePeer {
        public static int pickChoice(MJIEnv env, int objref, int limit) {
                int myChoice = 0;
                if (isFirstStepInsn()) {
                        log.fine("creating a choice generator");
                        ...
                } else {
                        IntChoiceFromSet cg =
                         getCurrentChoiceGenerator("probabilisticChoice",
                         IntChoiceFromSet.class);
                        myChoice = cg.getNextChoice();
                }
                return myChoice;
        }
}
```

Figure 5: (Simplified) Native Peer for the Choice Class

In AJPF, a specialised Probability Listener, executing in the NatJVM, listens for invocations of the `choose` method. The listener does not replace the code in `choose` but acquires a reference to the `Choice` object itself and after execution of the method completes, it can access the value stored in `thischoice_probability`. This allows the Listener in the NatJVM to annotate the edge created in the model by the choice generator with the appropriate probability, thus annotating the relevant branch with the probability of taking that transition. Similar specialised Listeners could be used to annotate branches with other information (e.g., actions, time estimates) were the system to be adapted for use with other more expressive model-checking systems.

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.

5.2 Translation to PRISM

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

1. First we initialise the model. We set it as a discrete time Markov chain (`dtmc`), list the numbers of all states and state the initial state (`0`), and list all the propositions in the property and 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, followed by "->".

(b) Iterate over all its outgoing edges. For each edge, we:

   i. Print out the probability of that edge being traversed.

   ii. Print out the state number and the values of the propositions in the property for the resulting state.

5.3 Case Studies

5.3.1 A Simple Unmanned Aircraft

As an example we consider a simple program based on [30] in which an autonomous unmanned aircraft (UA) must detect and avoid potential collisions. The unmanned aircraft's radar is only 90% reliable, so it does not always perform an 'evade' maneouvre when a collision is possible. The agent controlling the unmanned aircraft is implemented in GWENDOLEN which does not contain any probabilistic aspects. However the agent was executed within an environment model programmed in Java where the `Choice` class was used to represent the unreliability of the sensor when the agent requested incoming perceptions[8]. The code for this simple unmanned aircraft can be found in Appendix B.

The model is tracking two predicates: $\mathcal{P}(collision)$, which means *a potential collision is perceptible in the environment*, and $\mathcal{A}_{ua}evade$, which means *the last action performed was the unmanned aircraft agent taking an evade maneouvre*. In the construction of a Java environment to be used by an AIL it is necessary to provide a set of *percepts*. These form a list of predicates that are theoretically perceptible. Precisely because we wish to explore issues of an agent failing to perceive something, the property specification language allows these to be referred to separately from internal 'mental' states of the agent. In this instance $\mathcal{P}(collision)$ can be interpreted as meaning that in the environment a collision is going to occur irrespective of whether the agent has perceived this fact. This allows us to describe properties that capture the potential unreliability of sensors. 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 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 Figure 6 alongside the full model exported in the PRISM input language[9]. Figure 7 provides a brief outline of some key features of PRISM's property specification language, a fragment of PCTL [12]. Its full semantics can be found in [24].

We model-checked the above program in PRISM against the property

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

[8] We would also be able to investigate properties of BDI programming languages with probabilistic features, providing their AIL implementation used the `Choice` class — see Section 6.1.

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

AJPF Model

States:

```
3:  A(ua,evade()) = false;
    P(collision()) = false;
4:  A(ua,evade()) = false;
    P(collision()) = true;
5:  A(ua,evade()) = false;
    P(collision()) = true;
6:  A(ua,evade()) = true;
    P(collision()) = false;
7:  A(ua,evade()) = true;
    P(collision()) = false;
 ...
```

Edges:

```
0.9 ::: 3-->4
0.09999999999999998 ::: 3-->11
1.0 ::: 4-->5
1.0 ::: 5-->6
 ...
```

PRISM Model

```
dtmc

module jpfModel
  state : [0 ..15] init 0;
  auaevade: bool init false;
  pcollision: bool init false;
  [] state = 2 -> 1.0:(state'=3) & (auaevade'= false) & (pcollision'= false);
  [] state = 3 -> 0.9:(state'=4) & (auaevade'= false) & (pcollision'= true)
    + 0.09999999999999998:(state'=11) & (auaevade'= false) & (pcollision'= true);
  [] state = 4 -> 1.0:(state'=5) & (auaevade'= false) & (pcollision'= true);
  [] state = 5 -> 1.0:(state'=6) & (auaevade'= false) & (pcollision'= true);
  [] state = 6 -> 1.0:(state'=7) & (auaevade'= true) & (pcollision'= false);
   ...
endmodule
```

Figure 6: 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:

$$\begin{aligned} \phi &::= \quad \texttt{true} \mid \texttt{a} \mid \phi \wedge \phi \mid \neg\phi \mid \texttt{P}^{\bowtie p}[\psi] \\ \psi &::= \quad \bigcirc\phi \mid \phi \texttt{U}^{\leq k}\phi \mid \phi \texttt{U} \phi \end{aligned}$$

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 $\Box$ (always) and $\Diamond$ (eventually) to be defined (see Appendix D). $\texttt{P}$ 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 $\texttt{P}^{\geq 0.98}\psi$ evaluated at state s 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.

Figure 7: The PRISM Property Specification language

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

For comparison purposes we also model-checked the program in AJPF. Since AJPF does not support probabilistic reasoning we checked a different property:

$$\mathcal{B}_{ua}\, collision \rightarrow \Diamond \mathcal{A}_{ua} evade \tag{1}$$

i.e., that if the UA came to believe there would be a collision then it would eventually make an evade maneuver.

AJPF		AJPF outputing to PRISM	
Time	Memory	Time	Memory
3s	229MB	3s	360MB

PRISM itself, then took 1.8s to build and check a model from the file produced by AJPF.

5.3.2 A More Complex Unmanned Aircraft

The BDI agent program described in the previous section is quite basic: the BDI agent in control of the autonomous unmanned aircraft can only perform "cruise" and "detect/avoid" manoeuvres. In order to test the capabilities of the AJPF to PRISM translator, and to validate the PRISM models it generates, we used a more complex BDI agent program based on work described in [30]. The program described in Section 5.3.1 has one agent, UA (the autonomous unmanned aircraft's decision-making system), consisting of three GWENDOLEN plans (see Appendix B). The program described here contains two agents (one for the UA and one for an Air Traffic Control system — ATC) and a total of 22 plans divided between the two agents (see Appendix C).

In this more complex BDI agent program the unmanned aircraft begins on the ground (the airport ramp) at the start of its mission. The UA agent then requests clearance to taxi from the ATC agent. Clearance is either given or denied. If it is denied, the UA will repeatedly ask for taxi clearance until it receives permission to taxi. When the UA receives taxi clearance it directs the unmanned aircraft into the runway holding position, a position to the side of the runway where the aircraft waits until it has clearance from the ATC agent to manoeuvre onto the runway itself. Once in the runway holding position the UA will request permission to manoeuvre onto the runway ("line up"). Once clearance is given the unmanned aircraft manoeuvres onto the runway where it lines up ready for take-off. Once again, the UA requests clearance from air traffic control, this time to take-off. When take-off clearance is given, the UA agent directs the unmanned aircraft to take-off. Once in flight the UA may receive messages from a forward-looking infrared (FLIR) sensor system on-board the unmanned aircraft, which is modelled within a Java class representing the UA agent's environment. If the sensor detects that there is another aircraft approaching on a collision course, it informs the UA via a percept, 'collision', that the unmanned aircraft is on a collision course. Upon receiving this percept the UA directs the unmanned aircraft to perform an evasive manoeuvre using the action 'evade'. Finally, the UA will land when the navigation

subsystem (again, modelled within the Java class representing the UA agent's environment) indicates the destination has been reached by adding a percept, 'landing'. The full GWENDOLEN code for this example can be seen in Appendix C.

In this example the sensor is given an accuracy of 90%, which means that if there is another aircraft on a collision course, then the sensor will accurately determine that this is the case with a probability of 0.9. We were able to use the PRISM model generated by AJPF to determine that the following probability was, indeed, 0.9:

$$\mathtt{P}^{=?}\Box\neg(\Diamond(\mathcal{P}(collision) \wedge \neg(\Diamond\mathcal{A}_{ua}evade)))$$

This property expresses the probability that it is *never* the case that the possibility of collision is perceptible yet the UA fails to take evasive action. In other words, there is a probability of 0.9 of the UA taking evasive action, which we would expect as the environment model contains a faulty sensor which has an accuracy of 90%.

We can also verify the following property:

$$\mathtt{P}^{=?}\Box(\Diamond(\mathcal{P}(collision) \wedge \neg(\Diamond\mathcal{A}_{ua}evade)))$$

This property expresses the probability that it is always the case that the possibility of collision is perceptible yet the UA fails to take evasive action. PRISM calculates this probability to be 0.1, as would be expected by inspection of the model.

Therefore, these results validate the accuracy of the PRISM model generated from AJPF and verified using PRISM, at least for these properties.

5.3.3 Computational Resources

In Section 4.2 we compared the time taken by AJPF to verify a set of properties using (i) the JPF model checker, and (ii) the SPIN model checker. We were able to compare these timings as AJPF and SPIN were working on the same Kripke structure of the agent program and the outputs of both model checkers were a simple Boolean value indicating the presence of an error in the model. PRISM, in contrast to SPIN and AJPF, uses probabilistic timed automata instead of Kripke structures and returns a probability for each property verified.[10] Therefore it is not possible to compare JPF's performance to PRISM's performance, as both model checkers are fundamentally different, and JPF cannot be used to verify probabilistic models. However, we can compare the computational resources used by the two case studies presented in this section: the simple UA and the more complex UA. The computational resources used were as follows:

		Generation (AJPF)		Verification (PRISM)	
Model	States	Time (s)	Memory (MB)	Time (s)	Memory (MB)
Simple UA	16	1	235	1.3	0.24
Complex UA	42	3	427	1.3	0.24
Complex UA 2	98	6	486	1.3	0.26
Complex UA 3	200	10	486	1.3	0.28
Complex UA 4	408	19	488	1.3	0.31

[10] It should be noted that it is possible to use PRISM as a non-probabilistic model checker when using non-probabilistic models. It is also possible to receive a Boolean value as output, e.g., when checking that a resulting probability is within a certain range. However, the typical use of PRISM is to discover probabilities associated with probabilistic models.

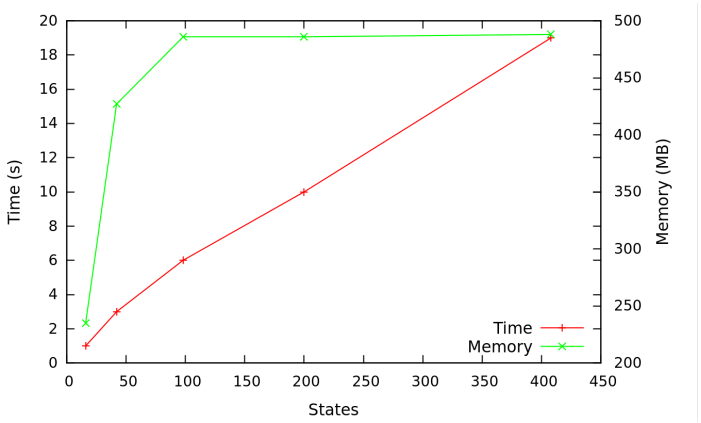

Figure 8: Time and memory resources used by AJPF for the generation of PRISM models.

These results are summarised in Figures 8 and 9. In addition to the simple and complex UA examples given earlier, we tested three further agent programs ("Complex UA [2-4]"). These models were extensions of the Complex UA agent program designed to increase the number of states required for model-checking. These modifications consisted of additional interactions with the agent's environment at the start of its execution. These results were obtained using an 8-core Intel Core i7-3720QM 2.60GHz CPU laptop with 16 GB of memory running 64-bit Ubuntu Linux 12.04 LTS.

In the table above, and in Figures 8 and 9, "States" refers to the number of states generated by AJPF and used in the PRISM model. The time and memory used for generation of the PRISM models by AJPF is shown under "Generation" and in Figure 8. The time and memory used for verification is shown under "Verification" and in Figure 9. It can be seen in Figure 8 that the amount of time used by AJPF to generate the models increases approximately linearly with the number of states. The memory used by AJPF for generation increased rapidly at first, but then levelled out, in line with typical AJPF usage. It can be seen in Figure 9 that the time used by PRISM during verification was constant[11], but the memory used increased with the number of states in an approximately linear fashion. However, the amount of memory used was minimal: in all cases less that 0.4 MB. PRISM's minimal overheads are not surprising given that it is an efficient symbolic model checker and therefore any time and memory used for such simple models should be similar.

In Section 4.2 we compared the efficiency of using AJPF with SPIN to using AJPF alone. In the case of PRISM verification, we cannot report a similar result as we could not compare verification times between (i) AJPF, and (ii) AJPF with PRISM, as AJPF does not support probabilistic model-checking. However, as in section 5.3.1, we could verify the program in AJPF alone against a similar but non-probabilistic property, (1) (see page 18). We show the time and memory consumption for this verification below.

[11]This is because the bulk of the time, when considering small models (i.e., those with a few hundred states) is taken up by system overheads which are the same for all runs.

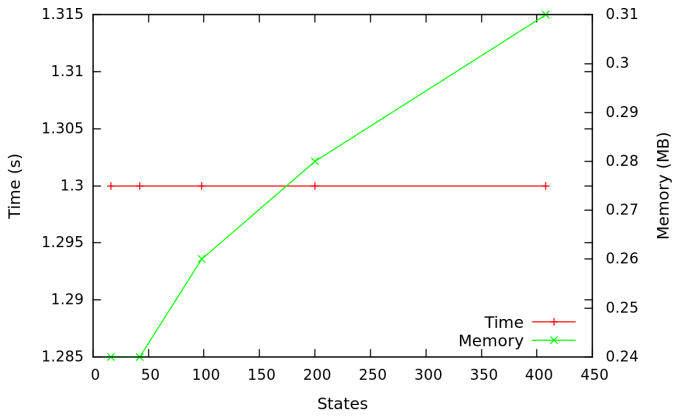

Figure 9: Time and memory resources used by PRISM for the verification of models generated by AJPF.

Model	States	Time (s)	Memory (MB)
Simple UA	4	1	235
Complex UA	33	3	235
Complex UA 2	81	4	297
Complex UA 3	167	7	429
Complex UA 4	343	13	305

However, clearly the advantage in using AJPF with PRISM over AJPF alone is precisely when we wish to verify properties that can not be expressed in AJPF; exporting models of agent programs from AJPF enables them to be model checked using probabilistic model checkers like PRISM. In principle it should be possible to export agent programs for other types of model checkers in order to model check agent programs in other ways. For instance, it may be possible to use AJPF to generate real-time agent program models (from a language such as AgentSpeak(RT) [28]) that could be model checked using a real-time model checker like UPPAAL [2]. Of course, this would depend on a real-time agent programming language interpreter being implemented using the AIL (see Section 2.1).

6 Conclusion

We have shown how AJPF can be used to generate models of BDI agent programs for formal verification using other model checkers in a two-step process. This work generalises the work of Hunter et. al [16], in which JPF was used to generate models of Brahms programs for model-checking using SPIN. The work described in the paper provides a generic tool for producing 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; the SPIN and PRISM model-checkers are used as

examples in this paper. 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. In both cases, this provides a viable, two-stage route to more flexible agent program verification.

6.1 Further Work

One of our primary motivations in performing this work was to enable the probabilistic model-checking of BDI agents, particularly in practical healthcare 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. The aim is to produce results about the overall reliability of systems based on probabilistic analyses of systems with sensors of varying reliability.

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 [23], 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 BDI programming languages that incorporate probabilistic features, something which will likely require that their AIL implementation uses the `Choice` class.

It would 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.

Finally, we aim to assess (and, hence, optimise) the model extraction process to (a) be as streamlined as possible, (b) produce structures that can potentially still take advantage of symbolic encodings in target model checkers, and (c) carry out simple abstractions, where appropriate. We will also explore the limits of this technique by identifying classes of programs that generate structures that are too complex to be verified using particular target model checkers.

Acknowledgments. The authors were partially funded by EPSRC projects EP/J011-770/1 (Reconfigurable Autonomy) and EP/K006193/1 (Trustworthy Robotic Assistants), and through the ERDF/NWDA-funded Virtual Engineering Centre.

References

[1] T. Babiak, M. Kretínský, V. Rehák, and J. Strejcek. 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*, pages 95–109. Springer, 2012.

[2] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, Oct. 1995.

[3] R. H. Bordini, M. Fisher, M. Wooldridge, and W. Visser. Property-based Slicing for Agent Verification. *J. Logic and Computation*, 19(6):1385–1425, Dec. 2009.

[4] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient Algorithms for the Verification of Temporal Properties. In *Formal Methods in System Design*, pages 275–288, 1992.

[5] L. A. Dennis and B. Farwer. Gwendolen: A BDI Language for Verifiable Agents. In *Proc. AISB Workshop on Logic and the Simulation of Interaction and Reasoning*. AISB, 2008.

[6] L. A. Dennis, M. Fisher, and M. Webster. Using Agent JPF to Build Models for Other Model Checkers. In *Proc. International Workshop on Computational Logic in Multi-Agent Systems (CLIMA)*, volume 8143 of *LNCS*, pages 273–289. Springer, 2013.

[7] L. A. Dennis, M. Fisher, M. Webster, and R. H. Bordini. Model Checking Agent Programming Languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[8] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier, 1990.

[9] M. Fisher, L. A. Dennis, and M. P. Webster. Verifying Autonomous Systems. *ACM Communications*, 56(9):84–93, September 2013.

[10] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. 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*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.

[11] E. C. Grigore, K. Eder, A. G. Pipe, C. Melhuish, and U. Leonards. Joint Action Understanding Improves Robot-to-Human Object Handover. In *Proc. IROS*, pages 4622–4629. IEEE, 2013.

[12] H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6:102–111, 1994.

[13] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. L. White. Formal Analysis of the Remote Agent Before and After Flight. In *Proc. 5th NASA Langley Formal Methods Workshop, Virginia, USA*, 2000.

[14] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Meyer. Agent Programming with Declarative Goals. In *Intelligent Agents VII*, volume 1986 of *LNAI*, pages 228–243. Springer, 2001.

[15] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[16] J. Hunter, F. Raimondi, N. Rungta, and R. Stocker. A Synergistic and Extensible Framework for Multi-Agent System Verification. In *Proc. 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 869–876. IFAAMAS, 2013.

[17] JPF... the Swiss Army Knife of Java^{TM} verification. `http://babelfish.arc.nasa.gov/trac/jpf/`. Accessed 2013-06-09.

[18] P. Kars. The Application of Promela and Spin in the BOS Project (Abstract), 1996. `http://spinroot.com/spin/Workshops/ws96/Ka.pdf`. Accessed 2013-05-30.

[19] M. T. Kirsch, V. A. Regenie, M. L. Aguilar, O. Gonzalez, M. Bay, M. L. Davis, C. H. Null, R. C. Scully, and R. A. Kichak. 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.

[20] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[21] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic Verification of Real-time Systems with Discrete Probability Distributions. *Theoretical Computer Science*, 282:101–150, 2002.

[22] N. K. Lincoln, S. M. Veres, L. A. Dennis, M. Fisher, and A. Lisitsa. Autonomous Asteroid Exploration — Agent Based Control for Autonomous Spacecraft in Complex Environments. *IEEE Computational Intelligence*, 8(4):25–38, 2013.

[23] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In *Proc. 21st International Conference on Computer Aided Verification (CAV)*, 2009.

[24] G. Norman, D. Parker, and J. Sproston. Model Checking for Probabilistic Timed Automata. *Formal Methods in System Design*, pages 1–27, 2012.

[25] PRISM: Probabilistic Symbolic Model Checker. `http://www.prismmodelchecker.org`. Accessed 2013-05-31.

[26] A. S. Rao and M. P. Georgeff. An Abstract Architecture for Rational Agents. In *Proc. International Conference on Knowledge Representation and Reasoning (KR&R)*, pages 439–449. Morgan Kaufmann, 1992.

[27] M. Sierhuis and W. J. Clancey. Modeling and Simulating Work Practice: A Method for Work Systems Design. *IEEE Intelligent Systems*, 17(5):32–41, 2002.

[28] K. Vikhorev, N. Alechina, R. Bordini, and B. Logan. An Operational Semantics for AgentSpeak(RT) (Preliminary Report). In *Proc. 9th International Workshop on Declarative Agent Languages and Technologies (DALT)*, pages 79–94, Taipei, Taiwan, May 2011.

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

[30] M. Webster, N. Cameron, M. Fisher, and M. Jump. Generating Certification Evidence for Autonomous Unmanned Aircraft Using Model Checking and Simulation. *Journal of Aerospace Information Systems*, 11(5):258–279, May 2014.

[31] M. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.

A The AJPF Property Specification Language

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:

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

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 [8] and are implemented as Büchi Automata as described in [10, 4]. Thus, the classical logic operators are defined by:

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

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:

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

The common temporal operators $\Diamond$ (eventually) and $\Box$ (always) are, in turn, derivable from $\mathcal{U}$ and $\mathcal{R}$ in the usual way [8].

B Simple Unmanned Aircraft Code

Syntax. GWENDOLEN uses many syntactic conventions from BDI agent languages: +!g indicates the addition of the goal g; +b indicates the addition of the belief b; while −b indicates the removal of the belief. Plans then consist of three parts, with the pattern

trigger : guard ← body;

The 'trigger' is typically the addition of a goal or a belief (beliefs may be acquired thanks to the operation of perception and as a result of internal deliberation); the 'guard' states conditions about the agent's beliefs (and, potentially, goals) which must be true before the plan can become active; and the 'body' is a stack of 'deeds' the agent performs in order to execute the plan. These deeds typically involve the addition and deletion of goals and beliefs as well as *actions* (e.g., evade) which refer to code that is delegated to non-rational parts of the systems.

The GWENDOLEN code for the simple unmanned aircraft case study is as follows:

```
:name: ua                                                                    1
: Initial Goals:                                                             2
fly [perform]                                                                3
:Plans:                                                                      4
+!fly [perform]: {⊤} ← +airborne;                                            5
+airborne: {⊤} ← normal, +normalFlight, +direction(straight);                6
+collision : {B airborne} ← evade;                                           7
```

C More Complex Unmanned Aircraft Code

The GWENDOLEN code for the more complex unmanned aircraft case study is as follows:

```
:name: ua
: Initial Beliefs:
waitingAtRamp
vicinityOfAerodrome
: Initial Goals:
fly [perform]
:Plans:
+.received(: tell , B): {⊤} ← +B;
+!fly [perform]: {B waitingAtRamp} ← .send(atc, :tell , requestingTaxiClearance);
+taxiClearanceDenied: {B waitingAtRamp} ← .send(atc, :tell, requestingTaxiClearance);
+taxiClearanceGiven: {B waitingAtRamp} ← −waitingAtRamp, +taxyingToRunwayHoldPosition;
+taxyingToRunwayHoldPosition: {⊤} ← −taxyingToRunwayHoldPosition, +holdingOnRunway;
+holdingOnRunway: {⊤} ← .send(atc, :tell, requestingLineUpClearance);
+lineUpClearanceGiven: {⊤} ← −holdingOnRunway, +linedUpOnRunway;
+linedUpOnRunway: {⊤} ← .send(atc, :tell, requestingTakeOffClearance);
+takeOffClearanceGiven: {⊤} ← −linedUpOnRunway, +takingOff;
+takingOff: {⊤} ← take_off , +airborne;
+airborne: {⊤} ← +normalFlight, normal, +direction( straight );
+collision : {B normalFlight, B direction(Dir)} ← −normalFlight,
        +oldstate(normalFlight),
        +olddir(Dir),
        +EmergencyAvoid,
        −direction(Dir), +direction( right ), evade;
+collision : {B changingHeading, B direction(Dir)} ← −changingHeading,
        +oldstate(changingHeading),
        +olddir(Dir),
        +EmergencyAvoid,
        −direction(Dir), +direction( right ), evade;
+changeHeading: {B normalFlight, ¯B toldOtherwise, B vicinityOfAerodrome} ← −normalFlight,
      +changingHeading, +direction(left);
+landing: {⊤} ← land;
+headingOK: {B changingHeading, B direction(Dir)} ← −changingHeading, +normalFlight,
      −direction(Dir), +direction(straight);
+objectPassed: {B emergencyAvoid, B oldstate(State), B olddir(Dir), B direction(D2)} ←
      −emergencyAvoid, −oldstate(State), +State, −olddir(Dir), −direction(D2), +direction(Dir);

:name: atc
: Initial Beliefs:
: Initial Goals:
:Plans:
+.received(: tell , B): {⊤} ← +B;
+requestingTaxiClearance: {⊤} ← .send(ua, : tell , taxiClearanceGiven);
+requestingTaxiClearance: {⊤} ← .send(ua, : tell , taxiClearanceDenied);
+requestingLineUpClearance: {⊤} ← .send(ua, : tell , lineUpClearanceGiven);
+requestingTakeOffClearance: {⊤} ← .send(ua, : tell , takeOffClearanceGiven);
```

D Syntax and Semantics of Temporal Logics

The syntax of Linear Temporal Logic is

$$\begin{array}{rl} \phi ::= & \texttt{true} \mid \texttt{false} \mid p \mid \neg\phi \mid \phi \wedge \phi \\ \psi ::= & \bigcirc\phi \mid \phi \texttt{U} \phi \end{array}$$

Where $p \in \Sigma$ such that Σ is a countable set of propositions.

Temporal formulae are interpreted over a discrete, linear model of time, $\mathcal{M}$ with an interpretation function, $I : \mathbb{N} \to 2^{\Sigma}$ which maps each point in time (represented as a natural number) to a set of propositions.

$$\langle \mathcal{M}, i \rangle \models p \quad \textbf{iff} \quad p \in I(i) \tag{2}$$
$$\langle \mathcal{M}, i \rangle \models \neg\phi \quad \textbf{iff} \quad \langle \mathcal{M}, i \rangle \not\models \phi \tag{3}$$
$$\langle \mathcal{M}, i \rangle \models \phi \wedge \psi \quad \textbf{iff} \quad \langle \mathcal{M}, i \rangle \models \phi \quad \textbf{and} \quad \langle \mathcal{M}, i \rangle \models \psi \tag{4}$$
$$\langle \mathcal{M}, i \rangle \models \bigcirc\phi \quad \textbf{iff} \quad \langle \mathcal{M}, i+1 \rangle \models \phi \tag{5}$$
$$\langle \mathcal{M}, i \rangle \models \phi \mathcal{U} \psi \quad \textbf{iff} \quad \exists j \geq i.\, \langle \mathcal{M}, j \rangle \models \psi \quad \textbf{and} \quad \forall i \leq k \leq j.\, \langle \mathcal{M}, k \rangle \models \phi \tag{6}$$

The syntax of probabilistic branching time logic, PCTL, is

$$\begin{array}{rl} \phi ::= & \texttt{true} \mid \texttt{a} \mid \phi \wedge \phi \mid \neg\phi \mid \texttt{P}^{\bowtie p}[\psi] \\ \psi ::= & \bigcirc\phi \mid \phi \texttt{U}^{\leq k} \phi \mid \phi \texttt{U} \phi \end{array}$$

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

We interpret PCTL formulae over discrete-time Markov chains (DTMCs) which capture a probabilistic branching model of time. Instead of evaluating a formula at a particular time point i as we did for LTL, we evaluate for a state s. Paths, π, through the DTMC are sequences of states $s_0(a_1, \mu_1)s_1(a_2, \mu_2)s_2 \ldots$, where a_i is the ith action taken to transform state s_{i-1} into state s_i and μ_i is the probability of that action occurring. $Paths(s)$ is the set of all paths that start in state s. The probability of a path, π is the product of the probability that each state in π transitions to the next state in π. The probability of a set of paths, Pr, is the sum of the probability of each individual path.

$$\langle \mathcal{M}, s \rangle \models p \quad \textbf{iff} \quad p \in I(s) \tag{7}$$
$$\langle \mathcal{M}, s \rangle \models \neg\phi \quad \textbf{iff} \quad \langle \mathcal{M}, s \rangle \not\models \phi \tag{8}$$
$$\langle \mathcal{M}, s \rangle \models \phi \wedge \psi \quad \textbf{iff} \quad \langle \mathcal{M}, s \rangle \models \phi \quad \textbf{and} \quad \langle \mathcal{M}, s \rangle \models \psi \tag{9}$$
$$\langle \mathcal{M}, \pi \rangle \models \bigcirc\phi \quad \textbf{iff} \quad \langle \mathcal{M}, s_1 \rangle \models \phi \tag{10}$$
$$\langle \mathcal{M}, \omega \rangle \models \phi \mathcal{U}^{\leq k} \psi \quad \textbf{iff} \quad \exists i \leq k.\, \langle \mathcal{M}, s_i \rangle \models \psi \,\textbf{and}\, \forall j < i.\, \langle \mathcal{M}, s_j \rangle \models \phi \tag{11}$$
$$\langle \mathcal{M}, \pi \rangle \models \phi \mathcal{U} \psi \quad \textbf{iff} \quad \exists k \geq 0.\, \langle \mathcal{M}, \pi \rangle \models \phi \mathcal{U}^{\leq k} \psi \tag{12}$$
$$\langle \mathcal{M}, s \rangle \models \texttt{P}^{\bowtie p}[\psi] \quad \textbf{iff} \quad Pr(\{\pi \in Paths(s) \mid \langle \mathcal{M}, \pi \rangle \models \psi\}) \bowtie p \tag{13}$$