Verifying Autonomous Systems*

Louise A. Dennis[1] [0000−0003−1426−1896]

University of Manchester, Manchester, UK, louise.dennis@manchester.ac.uk

Abstract. This paper focuses on the work of the Autonomy and Verification Network[1]. In particular it will look at the use of model-checking to verify the choices made by a cognitive agent in control of decision making within an autonomous system. It will consider the assumptions that need to be made about the environment in which the agent operates in order to perform that verification and how those assumptions can be validated via runtime monitoring. Lastly it will consider how compositional techniques can be used to combine the agent verification with verification of other components within the autonomous system.

Keywords: Verification · Autonomous Systems · Model-checking · Runtime Verification

1 Introduction

Autonomous systems are increasingly being used for a range of tasks from exploring dangerous environments, to assistance in our homes. If autonomous systems are to be deployed in such situations then their safety assurance (and certification) must be considered seriously.

Many people are seeking to leverage the power of machine learning to directly link inputs and outputs in a variety of autonomous systems via a statistical model. This paper examines an alternative, more modular, approach in which the decision making component of the system is constructed in a way that makes it amenable to formal verification. This approach necessitates an integrated approach to the verification of the whole autonomous system – both in terms of validating assumptions about the way the environment external to the system behaves and in terms of compositional verification of the various modules within the system.

2 A Cognitive Agent Decision Maker

Our decision making component is a cognitive agent programmed using the Beliefs-Desires-Intentions (BDI) programming paradigm.

At its most general, an *agent* is an abstract concept that represents an *autonomous* computational entity that makes its own decisions [39]. A general

* Supported by organization x.

[1] https://autonomy-and-verification.github.io

agent is simply the encapsulation of some computational component within a larger system. However, in many settings we desire something more transparent, where the reasons for choices can be inspected and analysed.

Cognitive agents [7, 33, 40] enable this kind of reasoning. We often describe a cognitive agent's *beliefs* and *goals*, which in turn determine the agent's *intentions*. Such agents make decisions about what action to perform, given their current beliefs, goals and intentions. This view of cognitive agents is encapsulated within the Beliefs-Desires-Intentions (BDI) model [32–34]. Beliefs represent the agent's (possibly incomplete, possibly incorrect) information about itself, other agents, and its environment, desires represent the agent's long-term goals while intentions represent the goals that the agent is actively pursuing.

There are *many* different agent programming languages and agent platforms based, at least in part, on the BDI approach [35, 5, 11, 29, 23]. Agents programmed in these languages commonly contain a set of *beliefs*, a set of *goals*, and a set of *plans*. Plans determine how an agent acts based on its beliefs and goals and form the basis for the selection of actions. As a result of executing a plan, the beliefs and goals of an agent may change and actions may be executed.

We consider agent architectures for autonomous systems in which a cognitive agent decision maker is supported by other components such as, image classifiers, sophisticated motion planning systems with statistical techniques for simultaneous localisation and mapping, planners and schedulers for determining when and in what order tasks should be performed, and health monitoring processes to determine if all the system components are functioning as they should. The agent decision-maker coordinates information and control between these systems.

3 Verifying Autonomous Choices

The starting point of our our approach is the use *formal verification* in the form of model-checking [10] (specifically, in our case, *program model-checking* [37]) for the cognitive agent.

Formal verification is the process of assessing whether a formal specification is satisfied on a formal description of a system. For a specific logical property, φ, there are many different approaches to this [21, 12, 6]. Model-checking takes a model of the system in question (or, in the case of program model-checking the implemented system itself), defining all the possible executions, and then checks a logical property against this model. Model-checking is therefore limited by our ability to characterise and feasibly explore all such executions.

The properties we verify are based on the *choices* the agent makes, given the information that is available to it. This is feasible since, while the space of possibilities covered by, for instance, the continuous dynamics of a robotic system is huge (and potentially infinite), the high-level decision-making within the agent typically involves reasoning within a discrete state space. The agent rarely, if ever, bases its choices directly on the *exact* values of sensors, etc. It might base its decision on values reaching a certain threshold, but relies on other parts of the system to alert it to this, and such alerts are typically binary valued

(either the threshold has been reached or it has not). We assume this information is transmitted to the agent in the form of *environmental predicates* which the agent then treats as beliefs.

A very simple example of this is shown in Figure 1. In this the agent decision maker uses two simple plans to choose whether to stop or follow a path. When it makes the choice it sends a command to an external control system (which can stop or execute path following behaviour). Information from sensors has been processed into two possible environmental predicates, `obstacle` or `path`. A property we might wish to verify here is

if the agent believes there is an obstacle then it will try to stop.

With only two predicates and this very simple behaviour we only need to explore four execution traces to see if the property is correct. The correctness will depend on the priority of the two plans. If their priority is incorrect then, in the case where the system detects both an obstacle and a path, it will follow the path rather than stopping. Errors of this kind, where priorities (or behaviour) are not correctly specified for situations where multiple events are taking place are typical of the errors we detect with our approach.

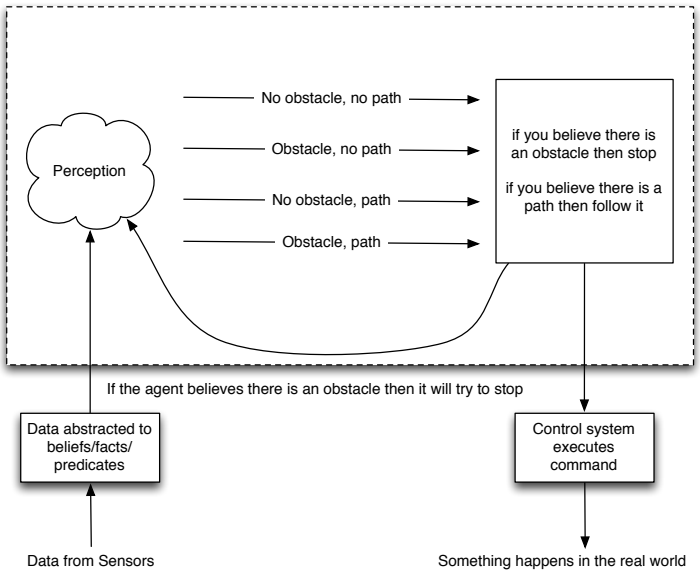

Fig. 1. Verifying a Simple Agent Decision Maker

3.1 The MCAPL Framework

We use the MCAPL framework [16, 13] in our work, which provides a route to the formal verification of cognitive agents and agent-based autonomous systems using program model-checking. The MCAPL framework has two main sub-components: the AIL-toolkit for implementing interpreters for BDI agent programming languages in Java and the AJPF model checker.

Agent JPF (AJPF) is a customisation of Java PathFinder (JPF) that is optimised for AIL-based language interpreters. JPF is an explicit-state open source model checker for Java programs [38, 28][2]. Agents programmed in languages that are implemented using the AIL-toolkit can thus be model checked in AJPF. Furthermore if they run in an environment programmed in Java, then the whole agent system can be model checked.

AJPF comes with a property specification for stating properties about BDI agent programs. This language is propositional linear temporal logic (PLTL), extended with specific modalities for checking the contents of the agent's belief base (**B**), goal base (**G**), actions taken (**A**) and intentions (**I**). This property specification language is described in [16]. As well as the modalities for beliefs etc,., the property specification language supports the standard logical connections ($\wedge$, $\vee$, $\neg$, $\implies$) and the temporal operators, $\Box$ (where $\Box p$ means that p is always true) and $\Diamond$ (where $\Diamond p$ means that p will eventually be true).

4 The Problem with Environments

In order to prove meaningful properties about our cognitive agent we need to consider the environmental predicates it receives from its environment and, importantly, sequences of these predicates as the situation in which the agent is operating evolves.

When we model check a decision-making agent in AJPF we *have to* use a purely Java environment to supply these predicates since JPF restricts us to the model-checking of Java programs. However in general agents controlling autonomous systems operate in a heterogenous environment with components programmed in a variety of languages and communicating via middleware such as the Robot Operating System [30] and behaviour ulitmately determined by the behaviour of the real world.

So when model-checking an autonomous hybrid agent system in AJPF we have to construct a Java *verification environment* that represents a simulation of some 'real' environment. We can encode assumptions about the behaviour of the 'real' world in this simulation, but we would prefer to minimize such assumptions. For much of our autonomous systems work we try to have minimal assumptions where on any given run of the program in the simulated environment, the environment asserts or retracts the environmental predicates that the agent receives on an entirely random basis. This means that we do not attempt to model assumptions about the effects an agent's actions may have on the

[2] https://github.com/javapathfinder

world, or assumptions about the sequence in which perceptions may appear to the agent. When model checking, the random behaviour of the verification environment causes the search tree to branch and the model checker to explore all environmental possibilities [15].

We call this most simple verification environment, where environmental predicates arrive at random, an *unstructured abstraction* of the world, as it makes no specific assumptions about the world behaviour and deals only with the possible incoming perceptions that the system may react to. Unstructured abstractions obviously lead to significant state space explosion so we have explored a number of ways to structure these abstractions in order to improve the efficiency of model checking, for example specifying that some predicates never appear at the same time. These *structured abstractions* of the world are grounded on assumptions that help prune the possible perceptions and hence control state space explosion.

What if these environmental assumptions turn out to be wrong?

Consider a simple intelligent cruise control programmed as a cognitive agent. This cruise control can perceive the environmental predicates `safe`, meaning it is safe to accelerate, `at_speed_limit`, meaning that the vehicle has reached its speed limit, `driver_brakes` and `driver_accelerates`, meaning that the driver is braking/accelerating. In order to formally verify the behaviour of the cruise control agent in an unstructured environment we would explore the behaviour for all subsets of {`safe`, `at_speed_limit`, `driver_brakes`, `driver_accelerates`} each time the vehicle takes an action. The generation of each subset causes the search space to branch so that, ultimately, all possible combinations, in all possible sequences of action are explored.

We would like to control the state space exploration by making assumptions about the environment. In the case of the cruise control, for instance, we might suggest that a car can never both brake and accelerate at the same time: subsets of environmental predicates containing both `driver_brakes` and `driver_accelerates` should not be supplied to the agent during verification, as they do not correspond to situations that we believe likely in the actual environment. However, since this introduces additional assumptions about environmental combinations it is important that we provide a mechanism for checking whether these assumptions are ever violated.

Runtime Verification We use a technology called *runtime verification* [36, 17] to monitor the environment that one of our autonomous systems finds itself in and check that this environment conforms to the assumptions used during verification. Our methodology is to verify the behaviour of the program using a structured abstraction prior to deployment – we refer to this as *static verification*. Then, during testing and after deployment, we continually check that the environment behaves as we expect. If it does not then the *runtime monitor* issues a violation signal. We do not discuss what should happen when a violation is detected but options include logging the violation for later analysis, handing over control to a human operator, or entering some fail-safe mode.

We can generate a verification environment for use by AJPF from a *trace expression*. Trace expressions are a specification formalism specifically designed for runtime verification and constrain the ways in which a stream of events may occur. The semantics of trace expressions is presented in [2]. A Prolog implementation exists which allows a system's developers to use trace expressions for runtime verification by automatically building a trace expression-driven monitor able to both observe events taking place and check their compliance with the expression. If the observed event is allowed in the current state – which is represented by a trace expression itself – it is consumed and a transition function generates a new trace expression representing the updated current state. If, on observing an event, no transition can be performed, the event is not allowed. In this situation an error is "thrown" by the monitor.

A trace expression specifying a verification environment can therefore be used in the actual *execution* environment to check that the real world behaves as the (structured) abstraction assumes. Essentially the verification environment represents a model of the real world and a runtime monitor can be used to check that the real world is behaving according to the model.

Figure 2 provides an overview of this system. A trace expression is used to generate a Java verification environment which is then used to verify an agent in AJPF (the dotted box on the right of the figure). Once this verification is successfully completed, the verified agent is used with an *abstraction engine* that converts sensor data into environmental predicates. This is shown in the dotted box on the left of the figure. If, at any point, the monitor observes an inconsistent event we can conclude that the real world is not behaving according to the assumptions used in the model during verification.

Verification Results We created trace expressions representing the property that the driver of a car only accelerates when it is safe to do so, and that the driver never presses both the brake and acceleration pedals at the same time.

From this trace expression we were able to generate a verification environment for the cruise control agent and compare it with performance on an unstructured abstraction. We chose to verify the property:

$$\Box(\mathbf{B}_{car}\,\mathsf{safe} \implies \Box(\Diamond(\mathbf{B}_{car}\,\mathsf{safe} \vee \mathbf{B}_{car}\,\mathsf{braking}))) \tag{1}$$

> *It is always the case that if the car believes it is safe (at some point) then it is always the case that eventually the car believes it is safe or the car believes it braking.*

We needed the initial $\mathbf{B}_{car}\,\mathsf{safe}$ in order to exclude those runs in which the car never believes it is safe since the braking behaviour is only triggered when the belief *safe* is removed.

When model checked using a typical hand-constructed unstructured abstraction, verification takes 4,906 states and 32:17 minutes to verify. Using the structured abstraction generated from the trace expression the property took 8:22

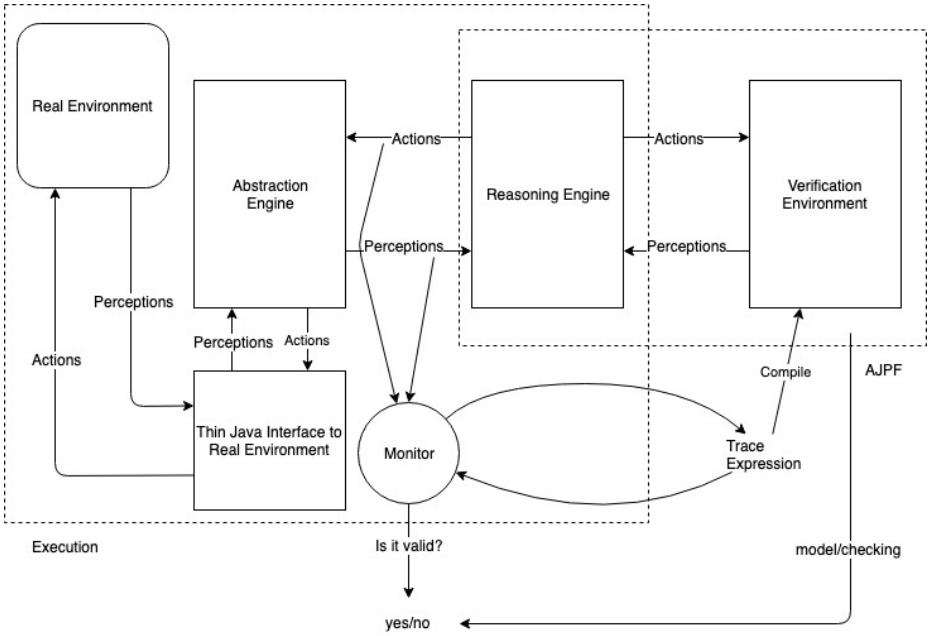

Fig. 2. General view of the runtime monitoring framework from [14]

minutes to prove using 1,677 states – this has more than halved the time and the state space.

As discussed, the structured abstraction may not reflect reality. In the original cruise control program the software could override the human driver if they attempted to accelerate in an unsafe situation. We removed this restriction. Now we had a version of the program that *was incorrect* with respect to our property in the unstructured environment model but remained correct in the structured environment model. We were then able to run our program in a motorway simulation contained in the MCAPL distribution where the "driver" could accelerate whenever they liked – the runtime monitor correctly detected the violation of the environment assumption and flagged up a warning.

Full details of the use of runtime verification with structured abstractions of environments can be found in [20].

5 Compositional Verification

We now look beyond our agent decision-maker to see how we can derive properties about the behaviour of an autonomous system of which the agent decision-maker is only one part. To motivate this we will discuss the verification of a vehicle platooning system (reported in [26]) and an autonomous search and rescue rover.

Vehicle Platooning The automotive industry is working on what are variously called *road trains*, *car convoys*, or *vehicle platoons*. Here, each vehicle autonomously follows the one in front of it, with the lead vehicle in the platoon/convoy/train being controlled by a human driver.

In these platoons, vehicle-to-vehicle (V2V) communication is used at the continuous control system level to adjust each vehicle's position in the lanes and the spacing between the vehicles. V2V is also used at higher levels, for example to communicate joining and leaving requests to the platoon's lead vehicle. It is these leaving and joining requests that we consider here.

We assume that this lead vehicle serves two roles. It is controlled directly by a human driver but it also acts as the central decision-making component in platoon operations such as leaving and joining protocols. Therefore there is a software agent in the lead vehicle, in all the other vehicles in the platoon and in any vehicle wishing to join the platoon. These software agents control the enactment of the protocols for joining and leaving the platoon.

Search and Rescue Rover Consider an autonomous rover deployed in a disaster situation. The autonomous rover has two types of sensor: the vision sensor is used for navigation around the area while an infrared sensor detects sources of heat that might indicate an injured or trapped person. There is a route planner that works with the vision system to provide obstacle free routes to target locations and a battery monitoring system that monitors the power level of the rover. Finally there are two cognitive agents: a goal reasoning agent which takes input from the sensors to select target locations for searching or recharging and a plan execution agent that selects and executes route plans based on battery level information and may send recharge instructions to the goal reasoning agent[3].

5.1 Module Level vs. System Level Properties

Vehicle Platoons We implemented the reasoning needed to follow the leaving and joining protocols for a platoon as a cognitive agent and were able to verify properties of these agents such as:

> *If a vehicle has a goal to join a platoon but never believes it has received confirmation from the leader, then it never initiates joining the platoon.*

However we are also interested in properties of the global system, for instance we might wish to know that

> *If an agent ever receives a joining agreement from the leader, then the cars in the platoon at the joining location have created enough space for the new car to join.*

[3] Code for these two agents can be found in the `src/examples/eass/compositional/rescue` directory of the MCAPL distribution.

In AJPF's property specification language this would involve both beliefs of the joining agent (that it has received an agreement) and actions of the agents in the specified places in the platoon (that space has opened up). We could, of course, verify each agent separately – for instance that the leader never *sends* an agreement message unless it believes that a space has opened but this fails to really tell us system behaviour. We also have a second problem. While it is all very well to verify that *eventually* an appropriate message is sent or an appropriate action is performed sometimes we require timing constraints on this – particularly in an environment such as a motorway where vehicles are moving at speed. So we are interested in properties like:

> *Given assumptions about the time taken for a vehicle to move to the correct spacing, to move to the correct place adjacent to the platoon and to change lane and for messages to be delivered then the time the required required for a completed join maneuver is within desired bounds*

AJPF's property specification language simply cannot express these properties.

Therefore we opted to use a different approach to verify global properties of the system. In this approach the agent is represented as a more simple automata with many implementation details and edge cases associated with handling unexpected environment behaviour, such as receiving unrequested agreements, removed. This simple automata is then combined with automata representing other vehicles and communication to verify timing properties using the UPPAAL model-checking tool [4, 3]. Meanwhile we use AJPF to prove desired properties of the agent implementation.

Search and Rescue Rover In the case of the Search and Rescue rover we are interested in verifying system level properties such as:

> *If the rover needs to recharge, it will execute actions to move to the charger location.*

This requires, at a minimum, formal guarantees about the behaviour of both agents and the route planning system, and ideally would also involve some analysis of the behaviour of the battery monitor.

In this case we can break this down into properties we want to hold of the individual system components and then combine these. For instance, we want to establish a couple of properties for the plan execution agent, namely:

> *If the plan execution agent believes the battery to be low and the current goal is not the charge position then it will send a recharge message to the goal agent.*

> *If the plan execution agent believes the current goal is the charge position and has a plan to get there then it will instruct the control system to follow the route to the charge position.*

We want to establish that the goal agent has the property:

If the goal agent believes a recharge is needed then it will set the target location to be the charge position.

The route-planner is not a BDI agent, but we can model the algorithm it uses and prove properties of that using Event-B [1]. For instance our route planner outputs a set of routes $\mathcal{R}$ as a sequence of waypoints, $w_0, \ldots, w_n$ so we established:

The current target location appears as a waypoint in all routes suggested by the route planner.

We then need a mechanism to combine these proofs.

5.2 Combining Results

In both our case studies we generated a number of formal verification results using different formalisms and technologies. The challenge is then to combine these into a system level result.

The Platoon For our platooning system, we established properties both of the agents controlling the individual vehicles involved in the platoon with details of the communication and control behaviour abstracted away in an unstructured verification environment and timing properties of the system behaviour with details of the agent behaviour abstracted away.

For simplicity, we assume that our system S consists of just two agents/vehicles and our verification works has given us the following:

- V_1 and V_2: timed automata representing the vehicle control used to verify properties in UppAal.
- V_1' and V_2': untimed abstractions of V_1 and V_2 represented in an unstructured verification environment in AJPF.
- A_1 and A_2: BDI agent implementations used to verify properties in AJPF.
- A_1' and A_2': abstractions of A_1 and A_2 with BDI elements removed used to verify properties in UppAal.
- *Comms12* is a timed automaton representing the inter-vehicle communications used to verify properties in UppAal.
- *Comms12*$'$ is an untimed abstraction of *Comms12* represented in an unstructured verification environment in AJPF.

We use $\|$ to represent the parallel combination of these automata into a system S. So $V_i' \parallel A_i \parallel$ *Comms12*$'$ represents a system used to prove a property about agent, A_i, in AJPF, while $V_1 \parallel A_1' \parallel$ *Comms12* $\parallel A_2' \parallel V_2$ is a system consisting of two agent abstractions and timed automata used to prove a property about interactions of the agents in UppAal. In [26] we prove that individual proofs about these systems containing abstractions can be conjoined into a single theorem about the system, $S = V_1 \parallel A_1 \parallel$ *Comms12* $\parallel A_2 \parallel V_2$.

We applied this to our platooning system. In AJPF we proved proved:

If a vehicle with a goal of joining the platoon never believes it has received confirmation from the leader, then it never initiates joining to the platoon.

While, in UppAal, we proved:

If an agent ever receives a joining agreement from the leader, then the preceding agent has increased its space to its front agent.

So the combined system has the property:

If a vehicle never believes it has received confirmation from the leader, then it never initiates joining to the platoon **and** *if an agent ever receives a joining agreement from the leader, then the preceding agent has increased its space to its front agent.*

Indicating that an agent never initiates joining the platoon unless the preceding agent has increased its space to it front agent.

Search and Rescue Rover In the platooning example, our combined property was expressed in a mixture of logics as used by the individual verification tools. For the search and rescue rover example we sought to place this kind of combination within a framework based on the concept of *contracts*.

For this system we specify contracts for each module, in the form of assumptions and guarantees and show, using first order logic, that these contracts imply the system properties. The verifications of individual modules allow us to argue that the module fulfils its contact.

Contracts in First-Order Logic We assume that our system consists of a set of modules, $\mathcal{M}$, and a signature, Σ, of variables.

For a given module, $C \in \mathcal{M}$, we specify its input modules, $\mathcal{I}_C \subseteq \mathcal{M}$, updates, $\mathcal{U}_C \subseteq \Sigma$, assumption, $\mathcal{A}_C : \Sigma \to \mathit{Bool}$ and guarantee, $\mathcal{G}_C : \Sigma \to \mathit{Bool}$. Taken together $\langle \mathcal{I}_C, \mathcal{U}_C, \mathcal{A}_C, \mathcal{G}_C \rangle$ form a *contract* for the module.

We use the notation $C^{\uparrow}$ to indicate that a C emits some output and $C^{\downarrow}$ to indicate that C receives an input.

We assume that all modules, C, obey the following:

$$\forall \phi, \overline{x} \cdot \overline{x} \subseteq \Sigma \backslash \mathcal{U}_C \wedge \mathcal{A}_C \wedge C^{\downarrow} \wedge \phi(\overline{x}) \Rightarrow \ \Diamond(\mathcal{G}_C \wedge C^{\uparrow} \wedge \phi(\overline{x})) \tag{2}$$

Intuitively, this states that if, at some point, C receives an input and its assumption holds then eventually it emits an output and its guarantee holds. Moreover, for any formula, ϕ, which does not involve any of C's update variables then, if ϕ holds when C recieves the input, ϕ also holds when C emits the output – i.e., ϕ is unaffected by the execution of C.

We have a second assumption that explains how inputs and outputs between two modules, C_1 and C_2, connect:

$$C_1^{\uparrow} \wedge C_1 \in \mathcal{I}_{C_2} \to C_2^{\downarrow} \tag{3}$$

Intuitively this states that if C_1 emits an output and is connected to the input of C_2, then C_2 recieves an input.

We use these two rules to reason about our system.

Module Contracts As an example module contract, the contract for the goal reasoning agent was:

Inputs $\mathcal{I}_G$: $\{V, H, E\}$
Updates $\mathcal{U}_G$: g
Assumption $\mathcal{A}_G$: $\top$
Guarantee $\mathcal{G}_G$: $(g \neq chargePos \Rightarrow$
$(\exists h \cdot h \in \mathbb{N} \wedge (g, h) \in GoalSet \wedge (\forall p, h_1 \cdot (p, h_1) \in GoalSet \Rightarrow h \geq h_1)))$
$\wedge (recharge \iff g = chargePos)$

The goal reasoning agent's inputs are the outputs of the Vision system V, the heat sensor, H and the plan execution agent, E. It updates the target goal, g. It has no assumptions ($\top$) and guarantees that:

1. If the target goal, g, (which it updates) is not the charge position then $(g, h) \in GoalSet$ for some heat signal, h, and for all other positions in the GoalSet the heat for that position is lower than h.
2. If a recharge is needed then g is the charge position

Does the goal reasoning agent meet its contract? We proved, using AJPF, that if the goal reasoning agent believed a recharge was required then it would set the goal to be the charging position. We also proved that if recharge was not required it would select the position with the highest heat signature. Note, however, we proved this for specific assumed positions (with these assumptions embedded in a verification environment), rather than the general properties stated in the contract.

Using our contracts we proved:

> *If at any point all plans sent to the plan execution agent by the planner module are longer than available battery power, then eventually the current plan will contain the charging position as the goal or there is no route to the charging position*

$$\Box(\mathcal{G}_P \wedge (\forall p \cdot p \in PlanSet \rightarrow length(p) > b - t) \wedge E^{\downarrow}) \implies \Diamond((g = chargePos \wedge g \in plan) \vee PlanSet \neq \varnothing) \quad (4)$$

using the two rules (3) and (2).

In a series of works [18, 9, 8] we have considered a number of variations on this example, as well as different kinds of contracts and sets of rules for reasoning about them.

6 Conclusion

This paper has focused on work performed by members of the Autonomy and Verification Network. In particular we have focused on the use of the MCAPL framework for verifying decision-making agents in autonomous systems [16, 15], the use of runtime verification to check that environments behave as assumed by abstractions [19, 20] and techniques for combining heterogenous verifications of different components or aspects of an autonomous system [26, 18, 9, 8].

Our approach is built upon constructing verifiable autonomous systems by locating key high-level decisions in a declarative component based upon the BDI-model of agency.

AJPF is not the only tool aimed at enabling the verification of autonomous systems. Tools are being developed for analysing the behaviour of image classifiers [25], reasoning about control systems [22], programming planning systems with defined formal properties [27] and validating both the models used by planning systems [31] and the plans produced [24]. This is why the work on compositional verification is so critical. To truly verify an autonomous system we need to consider all the software components that make up the system, verify each of them with the appropriate tools and then combine those verifications together.

Acknowledgements: This work has been supported by EPSRC, through Model-Checking Agent Programming Languages (EP/D052548), Engineering Autonomous Space Software (EP/F037201/1), Reconfigurable Autonomy (EP/J011770), Verifiable Autonomy (EP/L024845/1), Robotics and AI for Nuclear (EP/R026084/1), Future AI and Robotics for Space (EP/R026092/1), and the Trustworthy Autonomous Systems Verifiability Node (EP/V026801/1). Thanks are due to Rafael C. Cardoso, Marie Farrell, Angelo Ferrando, Michael Fisher, Maryam Kamali and Matthew Luckcuck for much of the work presented in this paper.

Data Access Statement The MCAPL framework, including most of the examples in this paper, is available on github, https://github.com/mcapl/mcapl, and archived at Zenodo, https://zenodo.org/record/5720861. The only example not available with the framework is the platooning example which can be found at https://github.com/VerifiableAutonomy/AgentPlatooning.

References

1. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
2. Ancona, D., Ferrando, A., Mascardi, V.: Comparing trace expressions and linear temporal logic for runtime verification. In: Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 9660, pp. 47–64. Springer (2016)
3. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Revised Lectures. Lecture Notes in Computer Science, vol. 3185, pp. 200–236. Springer (2004)

4. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In: Proc. of Workshop on Verification and Control of Hybrid Systems III. pp. 232–243. No. 1066 in Lecture Notes in Computer Science, Springer (Oct 1995)
5. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-agent Systems in AgentSpeak Using Jason. John Wiley & Sons (2007)
6. Boyer, R.S., Strother Moore, J. (eds.): The Correctness Problem in Computer Science. Academic Press (1981)
7. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press (1987)
8. Cardoso, R.C., Dennis, L.A., Farrell, M., Fisher, M., Luckcuck, M.: Towards compositional verification for modular robotic systems. In: Proc. 2nd International Workshop on Formal Methods for Autonomous Systems (FMAS 2020) (2020)
9. Cardoso, R.C., Farrell, M., Luckcuck, M., Ferrando, A., Fisher, M.: Heterogeneous verification of an autonomous curiosity rover. In: Proc. 12th International NASA Formal Methods Symposium (NFM). Lecture Notes in Computer Science, vol. 12229, pp. 353–360. Springer (2020)
10. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
11. Dastani, M., van Riemsdijk, M.B., Meyer, J.J.C.: Programming multi-agent systems in 3APL. chap. 2, pp. 39–67
12. DeMillo, R.A., Lipton, R.J., Perlis, A.: Social Processes and Proofs of Theorems of Programs. ACM Communications **22**(5), 271–280 (1979)
13. Dennis, L.A.: The MCAPL Framework including the Agent Infrastructure Layer and Agent Java Pathfinder. The Journal of Open Source Software **3**(24) (2018)
14. Dennis, L.A., Fisher, M.: Verifiable Autonomous Systems: Using Rational Agents to Provide Assurance about Decisions made by Machines. Cambridge University Press, in press
15. Dennis, L.A., Fisher, M., Lincoln, N.K., Lisitsa, A., Veres, S.M.: Practical Verification of Decision-Making in Agent-Based Autonomous Systems. Automated Software Engineering **23**(3), 305–359 (2016). https://doi.org/10.1007/s10515-014-0168-9, http://dx.doi.org/10.1007/s10515-014-0168-9
16. Dennis, L.A., Fisher, M., Webster, M., Bordini, R.H.: Model Checking Agent Programming Languages. Automated Software Engineering **19**(1), 5–63 (2012)
17. Falcone, Y., Havelund, K., Reger, G.: A Tutorial on Runtime Verification. In: Engineering Dependable Software Systems, pp. 141–175. IOS Press (2013)
18. Farrell, M., Cardoso, R.C., Dennis, L.A., Dixon, C., Fisher, M., Kourtis, G., Lisitsa, A., Luckcuck, M., Webster, M.: Modular verification of autonomous space robotics (2019)
19. Ferrando, A., Dennis, L.A., Ancona, D., Fisher, M., Mascardi, V.: Verifying and validating autonomous systems: Towards an integrated approach. In: Colombo, C., Leucker, M. (eds.) Runtime Verification. Lecture Notes in Computer Science, vol. 11237, pp. 263–281. Springer (2018)
20. Ferrando, A., Dennis, L.A., Cardoso, R.C., Fisher, M., Ancona, D., Mascardi, V.: Toward a holistic approach to verification and validation of autonomous cognitive systems. ACM Transactions on Software Engineering and Methodology **30**(4), 43:1–43:43 (2021). https://doi.org/10.1145/3447246, https://doi.org/10.1145/3447246
21. Fetzer, J.H.: Program Verification: The Very Idea. ACM Communications **31**(9), 1048–1063 (1988)
22. Garoche, P.L.: Formal Verification of Control System Software. Princeton University Press (2019), http://www.jstor.org/stable/j.ctv80cd4v

23. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming: Languages, Tools and Applications. pp. 119–157. Springer US, Boston, MA (2009)
24. Howey, R., Long, D., Fox, M.: VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning us ing PDDL. In: Proc. ICTAI. pp. 294–301 (2004). https://doi.org/10.1109/ICTAI.2004.120
25. Huang, X., Kroening, D., Ruan, W., Sharp, J., Sun, Y., Thamo, E., Wu, M., Yi, X.: A Survey of Safety and Trustworthiness of Deep Neural Networks: Verification, Testing, Adversarial Attack and Defence, and Interpretability. Computer Science Review **37**, 100270 (2020). https://doi.org/https://doi.org/10.1016/j.cosrev.2020.100270, http://www.sciencedirect.com/science/article/pii/S1574013719302527
26. Kamali, M., Dennis, L.A., McAree, O., Fisher, M., Veres, S.M.: Formal Verification of Autonomous Vehicle Platooning. Science of Computer Programming **148**, 88–106 (2017), http://arxiv.org/abs/1602.01718
27. Lacerda, B., Faruq, F., Parker, D., Hawes, N.: Probabilistic Planning with Formal Performance Guarantees for Mobile Service Robots. International Journal of Robotics Research **38**(9) (2019). https://doi.org/10.1177/0278364919856695, https://doi.org/10.1177/0278364919856695
28. Mehlitz, P.C., Rungta, N., Visser, W.: A Hands-on Java PathFinder Tutorial. In: Proc. 35th International Conference on Software Engineering (ICSE). pp. 1493–1495. IEEE / ACM (2013), http://dl.acm.org/citation.cfm?id=2486788
29. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. pp. 149–174
30. Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.Y.: ROS: An open-source robot operating system. In: Proc. ICRA Workshop on Open Source Software (2009)
31. Raimondi, F., Pecheur, C., Brat, G.: PDVer, a tool to verify PDDL planning domains. In: Proc. ICAPS'09 (2009), http://lvl.info.ucl.ac.be/Publications/PDVerAToolToVerifyPDDLPlanningDomains
32. Rao, A.S., Georgeff, M.P.: Modeling agents within a BDI-architecture. In: Proc. 2nd Int. Conf. Principles of Knowledge Representation and Reasoning (KR&R). pp. 473–484. Morgan Kaufmann (1991)
33. Rao, A.S., Georgeff, M.P.: An abstract architecture for rational agents. In: Proc. Int. Conf. Knowledge Representation and Reasoning (KR&R). pp. 439–449. Morgan Kaufmann (1992)
34. Rao, A.S., Georgeff, M.P.: BDI agents: From theory to practice. In: Proc. 1st Int. Conf. Multi-Agent Systems (ICMAS). pp. 312–319. San Francisco, USA (1995)
35. Rao, A.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Agents Breaking Away: Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World. LNCS, vol. 1038, pp. 42–55. Springer (1996)
36. Rosu, G., Havelund, K.: Rewriting-Based Techniques for Runtime Verification. Automated Software Engineering **12**(2), 151–197 (2005)
37. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (2003)
38. Visser, W., Mehlitz, P.C.: Model Checking Programs with Java PathFinder. In: Proc. 12th International SPIN Workshop. Lecture Notes in Computer Science, vol. 3639, p. 27. Springer (2005)
39. Wooldridge, M.: An Introduction to Multiagent Systems. John Wiley & Sons (2002)

40. Wooldridge, M., Rao, A. (eds.): Foundations of Rational Agency. Applied Logic Series, Kluwer Academic Publishers (1999)