Generating Certification Evidence for Autonomous Unmanned Aircraft Using Model Checking and Simulation

Matt Webster[1]
Department of Computer Science, University of Liverpool, Liverpool, UK

Neil Cameron[2]
Virtual Engineering Centre, STFC Daresbury Laboratory, Warrington, Cheshire, UK

Michael Fisher[3]
Department of Computer Science, University of Liverpool, Liverpool, UK

Mike Jump[4]
School of Engineering, University of Liverpool, Liverpool, UK

The use of unmanned aircraft for civil applications is expected to increase over the next decade, particularly in so-called "dull, dirty and dangerous" missions. Unmanned aircraft will undoubtedly require some form of autonomy in order to ensure safe operations for all airspace users. However, in order to be used for civil applications, unmanned aircraft must gain regulatory approval in a process known as "certification". This paper presents a proof-of-concept approach to the generation of certification evidence for autonomous unmanned aircraft based on a combination of formal verification and flight simulation. In particular, a class of autonomous systems controlled by rational agents is examined and we give examples of twenty-three different properties, based on the Rules of the Air and notions of Airmanship, which can be used in the formal model checking of rational agents controlling autonomous unmanned aircraft. Our techniques can be based on either (i) implicit models of the aircraft's physical environment specified in terms of the range of sensor inputs the autonomous system may receive, or (ii) more explicit physical models of the environment. Finally, we provide a description of how such formal verification can be used to refine the implementation of autonomous systems for unmanned aircraft.

Nomenclature

LTL	Linear Temporal Logic	$\mathsf{B}_a p$	Agent a believes p
$\Box p$	LTL: At all points in the future p is true	$\mathsf{G}_a p$	Agent a has a goal to do p
$\Diamond p$	LTL: At some point in the future p is true	$\mathsf{I}_a p$	Agent a intends to do p
$\neg p$	LTL: not p (i.e., p is not true)	$\mathsf{A}_a p$	Agent a does p
$\top$	LTL: Logical truth	$+x$	Agent adds the belief/goal x
$p \mathsf{U} q$	LTL: p is true until q is true	$-x$	Agent deletes the belief/goal x
$p \mathsf{R} q$	LTL: p releases q	COTS	Commercial Off-The-Shelf
$p \Longrightarrow q$	LTL: p implies q	Hz	Hertz
$p \wedge q$	LTL: p is true and q is true	km	Kilometres
$p \vee q$	LTL: p is true or q is true	UAS	Unmanned Aircraft System
$^\circ$	Degrees of arc	UI	User Interface

[1] Postdoctoral Research Associate, Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK. AIAA Member. Email: matt@liverpool.ac.uk.
[2] Postdoctoral Research Associate, Virtual Engineering Centre, STFC Daresbury Laboratory, Warrington WA4 4AD, UK.
[3] Professor, Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK.
[4] Lecturer, School of Engineering, University of Liverpool, Liverpool L69 3GH, UK.

I. Introduction

It is widely anticipated that the use of some form of unmanned aircraft in civil airspace will increase over the coming years [1, 2]. To be economically viable, unmanned aircraft will have to be operated in non-segregated airspace, i.e., airspace that is also available to other, more conventional users. To that end, unmanned aircraft will be subject to the same regulatory processes as manned aircraft and will therefore need to be certificated. During the certification process, evidence that a new aircraft type meets the regulations pertinent to its intended use is presented to a regulatory body, such as the Federal Aviation Administration (FAA) in the USA, or either the European Aviation Safety Agency (EASA) or the Civil Aviation Authority (CAA) in the UK. Once the regulator is satisfied that the regulations have been complied with, a *Type Certificate* will be issued for the new aircraft type, thus permitting its use within that country's civil airspace [1, 3].

The certification process for manned aircraft is well-understood as the current regulations have evolved over 100+ years of manned aviation. However, unmanned aircraft present a significant technological and regulatory challenge. While the elements of an unmanned aircraft that would be present on a manned aircraft can be certificated under existing regulations, those elements or functions that the regulations assume would be carried out by the on-board pilot must now also be certificated for the unmanned aircraft. This includes any software or hardware systems supporting a remotely-located pilot, or software/hardware systems designed to complement the remote pilot. This will have to be done using regulations that are still to be fully developed (and with far less experience than has been available for manned aviation). The certification guidelines (note *guidelines*, not regulations) that exist at present within the UK indicate that, to be certificated for use in civil airspace, any unmanned aircraft will need to be shown to be (i) no less safe than a manned aircraft, (ii) equivalent to manned aircraft in its behaviour, and (iii) transparent to other airspace users and existing infrastructure (for example, air traffic control) [4].

A key issue for unmanned aircraft is the use of autonomous systems, i.e., systems which have "the ability... to decide how to act so as to accomplish... delegated goals." (p. 23, [5]). Recent guidance published by the International Civil Aviation Organization (ICAO) has affirmed that, "In order for a UAS to operate in proximity to other civil aircraft, a remote pilot is... essential." [6] Often it is implied that the use of autonomous systems for aircraft is mutually exclusive with piloting, but this is not the case the majority of the time. Autonomous systems already have a long and successful history within manned aviation: autopilot and automated landing systems being two obvious examples. By definition, autonomous systems are given goals to achieve, and these goals must come from human operators. Furthermore, the use of autonomous systems can increase safety and operational effectiveness. For example, an autopilot is able to fly independently for long periods of time. This reduces the cognitive load on the pilot, who is then able to spend more time monitoring air traffic or maintaining other systems of the aircraft, for example. Remote piloting of unmanned aircraft creates additional challenges for pilots, particularly for situations of communications latency or loss, and is is likely that autonomous systems will be able to assist remote pilots in maintaining safe operations. For example, an autonomous system could maintain the current flight plan for a few seconds while communications are being re-established after an outage. Therefore, the use of autonomous systems on-board unmanned aircraft is compatible with, and indeed may be essential for, remote-piloting of those aircraft. In this paper we explore methods for verification of autonomous systems for use with unmanned aircraft. The systems in this paper do not explicitly mention a pilot, but the goals they are given to achieve and the parameters within which they act must all come from a pilot. Furthermore, it is assumed that the use of autonomous systems on unmanned aircraft is always accompanied by a "pilot override" function whereby a remote pilot can take control of the aircraft at any point.

The potential civilian uses of autonomous unmanned aircraft are manifold: remote sensing, disaster response, surveillance, search-and-rescue, to name but a few. However, while the military use of unmanned aircraft is increasing, the uptake in the civil sector has been much slower [1]. One reason for this is that potential manufacturers of autonomous unmanned aircraft, and the regulatory bodies tasked with their certification, are currently faced with a predicament. Manufacturers would like to have a definitive and recognised set of regulations in place before commiting to invest in the development and manufacture of a new autonomous unmanned aircraft. However, regulators appear reluctant to produce regulations when the full implications of the use of autonomous aircraft in civil airspace are unknown, as there are currently no such aircraft in routine use in non-segregated airspace.

It is our view that high fidelity *virtual prototypes* of autonomous unmanned aircraft operating within realistic simulations of civil airspace may provide part of the solution to this problem as they can be tested just as a real-life prototype would be, but at a fraction of the cost of real-world testing [7]. We also propose that formal verification tools such as model checking can be used to provide a level of assurance that an

autonomous system is behaving as it should: following the "Rules of the Air," and displaying airmanship where appropriate. These rules might relate, for example, to air traffic control clearance or emergency avoidance scenarios. [8] Such rules can be model checked using a model of the autonomous system's environment based on the information sent directly from sensors to the autonomous system. This environment model is implicit, i.e., the physical world is not modelled explicitly in terms of physical quantities, but rather in terms of how the sensor systems would perceive the physical world. However, some rules are more difficult to check in this way. For instance, certain rules pertain to numeric quantities, for example one Rule of the Air requires that an aircraft maintains a minimum separation of 500 feet from other aircraft [9].

(In fact, this rule pertains to "low flying" in the Rules of the Air. Other rules concerning safe separation state that "aircraft shall not be flown in such proximity to other aircraft as to create a danger of collision." We assume that a safe proximity would be at least 500 feet, and therefore the 500 feet rule will apply at all times during flight.)

In order to formally verify this rule by model checking, a model of an environment would need to be constructed in which complex concepts such as distance, time and separation are present. By using such an explicit (or, at least, *more* explicit) environmental model, quantitative Rules of the Air could indeed be formally verified using a model checker. Additionally, by integrating an already formally-verified autonomous unmanned aircraft virtual prototype within a distributed, networked, synthetic environment of the type described by Cameron et al. [10], the accuracy of the more explicit environment model can be assessed and verified.

Both of these approaches are investigated in this paper, and the way in which they might be used to gather evidence towards certification of unmanned aircraft is summarised in Fig. 1. Rational agent programs are developed to capture the autonomous decision-making within the UAS. These agent programs are compiled, formally verified using model checking and integrated within a virtual prototype of an autonomous unmanned aircraft. Results of multiple simulation runs can be analysed and the information gained can be used to refine (i) the environment model used during formal model checking, and (ii) the agent program.
The paper is structured as follows. In Section II the concept of a rational agent is introduced, and it is shown how a prototype autonomous agent-based control system for an unmanned aircraft was developed. The principles of model checking are also introduced here. In Section III it is shown how model checking, specifically developed for agent programs, can be used in the formal verification of agent control for autonomous unmanned aircraft. In Section IV the development of a higher fidelity environment model for agent model checking is considered, based on physical models of the civil airspace environment. This enables quantitative requirements (for example, certain Rules of the Air) to be formally verified. In particular, it is formally verified that an agent-based control system for an autonomous unmanned aircraft will always maintain at least 500 feet separation from an intruder aircraft. In Section V it is shown how agent programs for model checking can be debugged and improved using formal verification. Finally, in Section VI, it is described how the combination of formal model checking and simulation might be used to generate evidence that might contribute towards the certification of autonomous unmanned aircraft. Comparisons are made with similar work in the literature and directions for future research are provided.

This paper is based on earlier works [8, 11] in which a rational agent-based autonomous control system was model checked with respect to the Rules of the Air and using a higher fidelity environment model. This paper expands on these works as follows:

1. In [8] we describe how a relatively simple model of an autonomous unmanned aircraft can be developed in PROMELA[12] and Java [13] and model checked using the SPIN[12] and Agent JPF [14] model checkers respectively. In each case the model is checked exhaustively relative to three properties based on a small subset of the Rules of the Air. The model based on Java was found to be more fruitful as the Java implementation allowed an easier integration into a real-time high fidelity flight simulation laboratory, in which the rational agent in control of an unmanned aircraft could be developed and analysed in a variety of realistic scenarios within civil airspace. This integration work was described by Cameron et al [10]. In this paper we take this more fruitful model and verify it against a larger subset of the Rules of the Air. In addition, we verify this model against a set of 15 airmanship requirements derived from interviews with subject matter experts. These requirements are described in detail in Section III.

2. In [11] the idea of model checking a rational agent against a higher fidelity environment model was introduced and it was shown that this could be used to model check a Rule of the Air concerning minimum safe separation from other airspace users. In this paper we describe this approach in greater

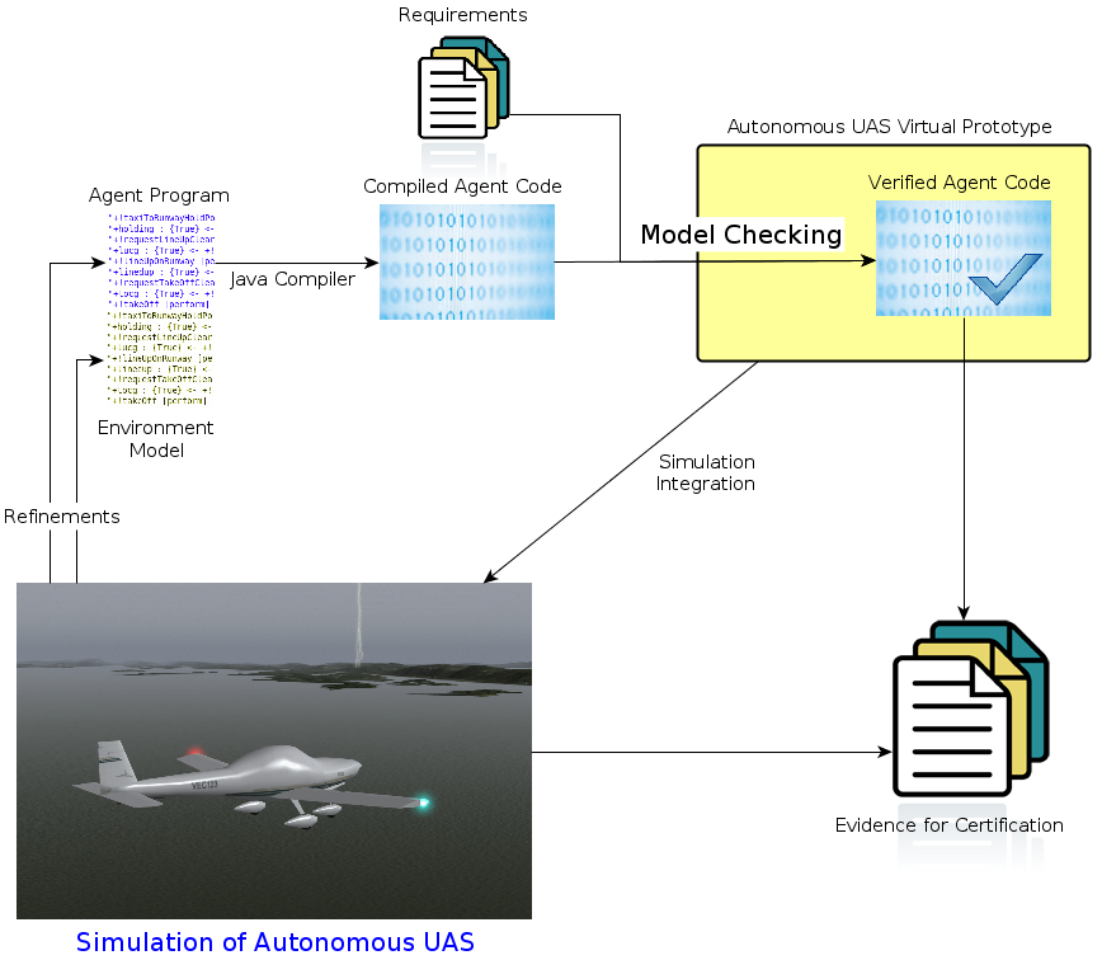


Fig. 1 Using model checking and simulation to gather evidence for certification of autonomous unmanned aircraft.

detail, provide additional information on the revised higher fidelity environment model (e.g., Figure 8) and in Section IV D discuss the trade-offs involved in using a higher fidelity environment model versus lower fidelity, more abstract, environment models.

3. In Section V it is shown how model checking can be used as a tool for guiding the development of a software system using an iterative design–implement–verify development process. We show how one of the formalised requirements from Section III was found to be false by the model checker for a version of the Executive. We describe the process for tracking down the source of this error, show how it was corrected, and describe how the Executive was then model checked again in order to verify this requirement.

Furthermore, it is intended that the consolidation of the earlier work with the new developments described above will provide a more detailed and comprehensive description of our approach to providing evidence for certification of autonomous unmanned aircraft.

II. Background

A. Rational Agents

An agent is a computer system, situated within an environment, that is capable of *autonomous* action within that environment towards its objectives. We take "autonomy" to mean "the ability and requirement to decide how to act so as to accomplish. . . delegated goals." (p. 23, [5]) In other words, an autonomous system is a system which is capable of acting independently towards its goals. Agent systems are (by definition) autonomous systems, and have been used for numerous applications in which autonomous behaviours are

desirable, including distributed sensing, electronic commerce and social simulation [5], as well as safety- and mission-critical applications (see, e.g., [15–18]).

In the work described in this paper, agent systems are used to manage and control the safe flight of a virtual prototype autonomous unmanned aircraft operating in a virtual civil airspace environment. The agent architecture used in this paper is based on the Belief–Desire–Intention (BDI) model of *rational* agent reasoning [19], in which the agent has beliefs about its environment, together with desires (things which it would like to become true within its environment) and intentions (the particular deeds that the agent deems necessary to achieve its desires based on its beliefs). It is important to point out that the BDI architecture is metaphorical; it is not suggested that agent systems have beliefs, desires or intentions in the full human cognitive sense. Rather, beliefs, desires and intentions are used as abstractions for capturing autonomous behaviours. People tend to think in terms of beliefs, desires and intentions, and so this is a natural and expressive way to encode autonomous behaviours. Furthermore, use of the BDI architecture results in rational agents whose behaviour can be usefully explained in terms of rationality and reasoning. Crucially, for certification purposes, the reasons for high-level decisions in rational agents are explicit and explainable.

Gwendolen [20], a BDI agent programming language, is the implementation language we use to program abstract behaviours for an autonomous unmanned aircraft control system. Gwendolen agents are rational agents composed of the following components:

- A *name* for the agent.
- An *environment* in which the agent executes. Every agent requires some environment in which it executes. The environment may send information to the agent and may respond to agents' actions. As Gwendolen is based on Java [13] the environment for the agent is defined as a Java class.
- A set of *initial beliefs*. These define what the agent believes at the beginning of its execution. Beliefs are structured as ground first-order logic predicates [21], for example "stopped", "altitude(1000)" and "route(London(lhr), NewYork(jfk), etd(1230), eta(0130))" could all be beliefs.
- A set of *initial goals*. These define what the agent desires at the beginning of its execution. Note that in Gwendolen (as in the BDI approach) "goals" are the name given for desires. There are three kinds of goals: "perform goals", where the agent must perform some act, "achieve goals", in which the agent must achieve something, and "maintain goals", in which the agent must maintain something. Goals are also structured as first-order logic predicates, with an additional annotation showing the type of the goals, e.g., $!_a g$ is goal to achieve g, $!_p h$ is a goal to perform h and $!_m i$ is a goal to maintain i.
- A set of *plans*. Gwendolen plans have the form:

  `event : guard <- deeds`

  Each plan begins with a triggering `event`, such as the addition of a belief about sensor data or a message sent from another agent. Next is the `guard`, which consists of a set of terms which may be true or false. Typically these terms are about the agent's beliefs and goals, for example "the agent a believes that its altitude is 10,000 feet" and "the agent a has a goal to land at airport N" can be encoded as ground terms $\mathsf{B_a}(\text{altitude}(10000))$ and $\mathsf{G_a}(\text{land}(\text{N}))$ respectively. If a triggering event occurs and the statements in the guard are found to be true, then the `deeds` in the plan are enacted by the agent. The deeds can include actions within the agent's environment, sending messages to other agents, adding/removing beliefs in the agent's own belief database, and so on. For example:

  $+!_a$ *missionComplete* : {*B flightPhase* (*unknown*), $\neg$*B connectionOk*} <− *send*(*env*,*connect*); 1

  This means that if the achieve-goal *missionComplete* is added, and if we believe that the flight phase is unknown, and we do not believe that the connection to the environment is "okay", then send a message ("connect") to the environment.

The full syntax and semantics for the Gwendolen agent programming language is given by Dennis & Farwer in [20] while further examples of Gwendolen programs can be found in [14]. We provide an example of Gwendolen agent code used in our system in Fig. 4.

B. Agents for Unmanned Aircraft

Rational agents are well-suited for use in autonomous unmanned aircraft, as they can provide the overall direction and control for a task or mission, and their behaviour can be explained by analysing the beliefs, goals, intentions and plans which define their behaviour. A generic architecture for the use of agents within unmanned aircraft (and autonomous systems in general) is given in Fig. 2. Information flows into the flight control system (FCS) and rational agent from the environment. Both components communicate, with the rational agent making abstract decisions about the progress of the mission (when to taxi, when and what to communicate with air traffic control, where to fly, etc.). These abstract decisions are then passed to the flight control system which determines the exact low-level inputs to actuators for the aircraft's subsystems (e.g., engine, control surfaces, etc.) This generic architecture can of course be used beyond unmanned aircraft, for example in ground vehicles, spacecraft, marine vehicles, robotics, etc. [22, 23]

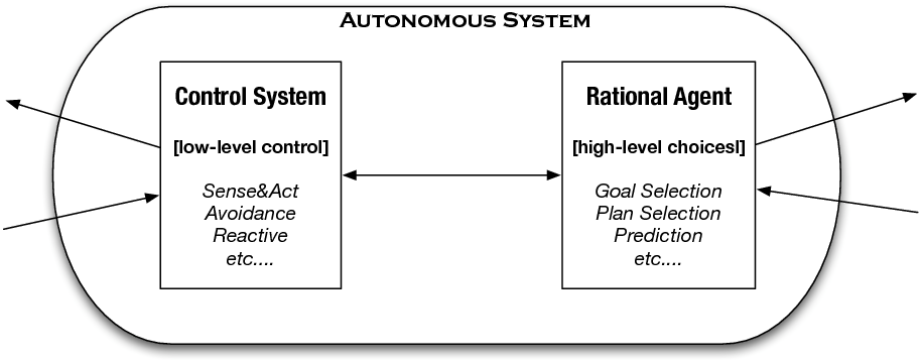


Fig. 2 A generic architecture for rational agent-based autonomous systems.

A proof-of-concept virtual prototype unmanned aircraft system was developed using this architecture; given that the work reported here was conducted as part of a wider Virtual Engineering Centre (VEC) [24] project, the vehicle was given the registration "VEC123". It will be referred to as such throughout the remainder of the paper. The rational agent in overall control, known as the *Executive*, was written in Gwendolen and connected via network socket interfaces to the flight control system. The Executive comprises a set of plans which contribute to the successful completion of its missions and is capable of controlling the autonomous flight of a virtual prototype unmanned aircraft through simulated civil airspace whilst observing a subset of the published "Rules of the Air". Missions are, in general, divided up into a number of flight phases representing the behaviour of the vehicle at different points of its mission:

1. Waiting at the airport ramp.
2. Taxying.
3. Holding at runway hold position prior to line-up.
4. Lining up on runway prior to take off.
5. Lined up and ready for take off.
6. Take off.
7. Cruise.
8. Emergency avoid.
9. Aerodrome approach.
10. Landing and stopping.
11. Taxying from runway to ramp (i.e., taxying at destination).

Based on our requirements, it was determined that VEC123's Executive would need to communicate with a range of different subsystems: air traffic control communication devices, fuel sensors, route planners,

detect-and-avoid sensors, etc. After assessing alternatives a "polling" architecture was developed in which the Executive polls each subsystem in turn in order to avoid the various subsystems overwhelming the Executive with information. In this way the Executive is able to control which of its subsystems it is interacting with at any given point. This has the effect of reducing the possibility of the Executive agent becoming incapacitated due to information overload, as well as reducing the size of the Executive agent's state space. (The relevance of the latter will become apparent later.) The polling architecture of the Executive and its subsystems is shown in Fig. 3. The following subsystems are included in the virtual prototype at present.

- *ATC* is the air traffic control communication subsystem, responsible for communication with air traffic control during all phases of flight.
- *DAS* is the detect-and-avoid sensor, and is responsible for the detection of nearby aircraft that pose a hazard to safe operations (known as "intruder aircraft").
- *Env* is the environment reporting subsystem, and is responsible for providing the Executive with information on various aspects of the unmanned aircraft's environment, including inclement weather, regions of closed airspace, changed rules at aerodromes, and other notices.
- *Fuel* is the fuel subsystem, and is responsible for providing information on the fuel state of the aircraft.
- *Nav* is the navigation subsystem, and is responsible for providing information on the location of the unmanned aircraft.
- *Planner* is the route planner, and is responsible for generating detailed route plans to navigate through civil airspace. Typically, the Planner is provided with a current location, an intended destination and a fuel level, and will generate an abstract description of a route plan which is returned to the Executive agent for inspection.
- *Veh* is the vehicle subsystem, responsible for providing information on the vehicle's system health (for example, engine operation status) and maintaining the vehicle's flight phase (for example, waiting at ramp, cruise, approach, etc.).

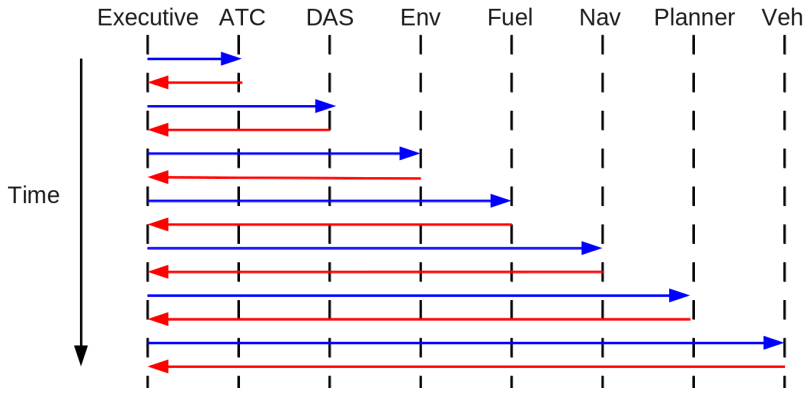


Fig. 3 Sequence diagram showing the polling architecture of the Executive's communication with its subsystems. The Executive polls its subsystems repeatedly using the above pattern. The rightward arrows represent poll messages and the leftward arrows represent response messages.

Typically, the Executive's beliefs are formed as a result of sensor inputs via the subsystems above, which are distilled from continuous "real-world" data. For example, a sensor system might relay that, "There is an aircraft 541.2 feet away travelling on a heading of 220.8° at a speed of 161.3 knots." This could be stored precisely, for example $aircraft(541.2, 220.8, 161.3)$, or stored more abstractly, for example $aircraft(540, 220, 160)$ or even as $safeSeparationFrom(aircraft)$. During the agent's reasoning process it may decide to take some action which is then forwarded on to the actuators that form part of the control system of the unmanned aircraft. For instance, an action $turnToHeading(0)$ could map to a command to the heading control system to alter course until VEC123's heading is equal to 0°.

VEC123 is designed with a modular, extensible architecture so that different sensor models can be used in place of each subsystem, and sensor models of different fidelity can be examined to see how they affect the behaviour of the unmanned aircraft. In addition, subsystems can be added or removed depending on the scenario being examined. For example, we may wish to consider a scenario in which communication from air traffic control is being disrupted. In this case we could remove the ATC subsystem to examine the behaviour of VEC123 without information from ATC. Alternatively, we may wish to adapt VEC123 for use in a radioactive environment, in which case we can add a particle detector subsystem for measuring radiation levels. Furthermore, different versions of the Executive can be used in order to examine different behaviours of the unmanned aircraft. For the purposes of model checking in Section III, we assume that sensors are always accurate (i.e., sensors have access to truth data within the simulation). It is possible to use model checking to examine situations where sensors are not always completely accurate, e.g., [25], but this is beyond the immediate scope of this work.

The Executive's plans are designed to direct the autonomous flight of VEC123 through its flight phases towards successful mission completion. A subset of the Executive's plans is shown in Fig. 4. The first plan indicates that when the agent receives a *tell* message from another agent, the agent will add the content of that message (B) as a new belief. This happens unconditionally, as the guard is true ($\top$). The second plan indicates that if the agent gets a new perform-goal, *startup*, then it will add the achieve-goal *missionComplete*. The third plan indicates that when the achieve-goal *missionComplete* is added, and if the agent believes that the flight phase is unknown, and it doesn't believe that the connection to the environment is *Ok*, then the agent will send a message to the environment in order to connect. The fourth plan indicates that if the agent gets a new belief that the connection is *Ok*, and it is doesn't believe that it has sent a message to the "Veh" subsystem to request flight phase (*reqfp*), and it believes that its flight phase is unknown, then it will add a belief that the agent has polled itself. This starts the polling mechanism of the agent as described above. (This plan also shows the use of "lock" and "unlock", which together cause the agent to execute all of the deeds as a single atomic deed.) The remaining plans request information about the unmanned aircraft's flight phase from the Vehicle subsystem (line 5), determine the fuel level from the Fuel subsystem (line 6), determine the current position from the Nav subsystem (line 8), request taxi clearance from air traffic control via the ATC subsystem (line 10), and request a route for taxying from the Planner subsystem (line 12). Lines 7, 9 and 11 make the Executive agent continue polling all of its subsystems for information while it is waiting for information from a specific subsystem.

+. *received* (: *tell* , *B*): {$\top$} <– +*B*; 1

+!$_p$*startup* : {$\top$} <– +!$_a$*missionComplete*; 2

+!$_a$*missionComplete* : {*B flightPhase* (*unknown*), ¬*B connectionOk*} <– *send*(*env*,*connect*); 3

+*connectionOk* : {¬*B sent*(*veh*, *reqfp*), *B flightPhase* (*unknown*)} <– lock, +*polled*(*self*), unlock; 4

+*poll*(*self*) : {¬*B sent*(*veh*, *reqfp*), *B flightPhase* (*unknown*)} <– lock, +*sent*(*veh*, *reqfp*), *send*(*veh*, *reqfp*), 5
−*poll*(*self*), +*polled*(*self*), unlock;

+*poll*(*self*) : {*B veh*(*status* ,*waitingAtRamp*), *B flightPhase* (*unknown*), *B sent* (*veh*, *reqfp*)} <– lock, 6
−*veh*(*status* ,*waitingAtRamp*), −*poll*(*self*), −*sent*(*veh*, *reqfp*), −*flightPhase*(*unknown*),
+*flightPhase* (*waitingAtRamp*), *send*(*fuel* , *reqfuelstatus*), +*waiting*(*fuel* , *reqfuelstatus*), +*polled*(*self*),
unlock;

+*poll*(*self*) : {¬*B veh*(*status* ,*X*), *B flightPhase* (*unknown*), *B sent* (*veh*, *reqfp*)} <– lock, −*poll*(*self*), 7
+*polled*(*self*), unlock;

+*poll*(*self*) : { *B waiting* (*fuel* , *reqfuelstatus*), *B fuel* (*level* ,*F*) } <– lock, −*poll*(*self*), 8
−*waiting*(*fuel* , *reqfuelstatus*), *send*(*nav*, *reqposition*), +*waiting*(*nav*, *reqposition*), +*polled*(*self*), unlock
;

+*poll*(*self*) : { *B waiting* (*fuel* , *reqfuelstatus*), ¬*B fuel* (*level* ,*F*) } <– lock, −*poll*(*self*), +*polled*(*self*), 9
unlock;

+*poll*(*self*) : {*B position* (*Lat*,*Lon*,*Alt*), *B fuel* (*level* ,*F*), *B waiting* (*nav*, *reqposition*)} <– lock, 10
−*poll*(*self*), −*waiting*(*nav*, *reqposition*), *send*(*atc* , *reqtc*), +*waiting*(*atc* , *tc*), +*polled*(*self*), unlock;

+*poll*(*self*) : {¬*B position* (*Lat*,*Lon*,*Alt*), *B fuel* (*level* ,*F*), *B destination* (*D*), *B waiting* (*nav*, *reqposition*)} 11
<– lock, −*poll*(*self*), +*polled*(*self*), unlock;

+*poll*(*self*) : {*B flightPhase* (*waitingAtRamp*), *B atc*(*tcg* ,*R*), *B waiting* (*atc* , *tc*), *B position* (*Lat*,*Lon*,*Alt*), *B* 12
fuel (*level* ,*F*)} <– lock, −*poll*(*self*), −*waiting*(*atc* , *tc*), *send*(*planner* , *reqTaxiRoute* ,*R*,*Lat* ,*Lon*,*F*),
+*waiting*(*planner* , *reqTaxiRoute*), +*polled*(*self*), unlock;

Fig. 4 A subset of the Executive agent's plans, written in Gwendolen.

C. Formal Methods and Model Checking

Formal methods comprise a family of techniques for ensuring that systems (typically, programs) will work as intended. Formal methods involve formal verification, in which mathematical/logical techniques are used to verify computer systems. Model checking is one such approach which uses an exhaustive exploration of the state space of a program or protocol [26, 27]. Model checking was first developed to verify the behaviour of communication protocols yet, increasingly, programs are being verified using model checking (e.g., [28]).

As the name suggests, *model checking* involves exhaustively analysing a model describing all the program/system's possible executions. This model is a mathematical structure and it is typically assessed against some requirements provided in formal logic [26]. A model checker will automatically assess each potential execution and, if it finds any such that violates the required formulae, will notify the user.

In many cases, the construction of a separate mathematical model of all the possible system behaviours is both expensive and inconvenient. An alternative, called *program* model checking [28] works directly on the program rather than building a model for it. This approach works by executing the program within a sandbox-type environment. (A simplified description of how program model checking works is as follows; see [26, 29] for more detailed descriptions.) At each state in the program or process, the model checker checks whether a given property holds. If the property holds, then the model checker continues on to the next state. If the property does not hold, then the model checker prints out an error message and terminates. If the model checker reaches the end state of the program without terminating then the property has been satisfied in all states of the program, and the program can be considered to be verified for that property. The model checker will then print out a success message and terminate. Whenever a branching point is reached, that is, a point at which two or more different things could potentially happen, the model checker explores each branch. Programs can, of course, have many branches, and this gives rise to the main limiting factor when using model checkers, known as the "state space explosion". As the number of branches increases, the number of ways of executing the program increases exponentially, resulting in much longer execution times for the model checker.

One such program checker is Java PathFinder (JPF), developed at NASA Ames Research Center for model checking Java programs [28]. Program checkers like Java PathFinder take much longer to verify a program than typical model checkers. Model checkers work on the order of seconds to verify programs [12], whereas program checkers can take much longer: hours, minutes or even days [14]. Fortunately, many programs are deterministic. As deterministic programs execute in the same way every time, there are no branching points. However, non-determinism often arises through the uncertainty in the program's environment. For example, in our case, a chief source of non-determinism is Java's multithreading combined with explicit non-determinism in the model of the agent's environment.

Java PathFinder forms the basis of Agent JPF, the first agent program checker for a variety of Java-based agent languages [14, 30]. As Agent JPF is based on Java, agent programs are executed by passing the agent programs as an input to a Java program. The property to be verified is also given as an input to a Java program and is encoded in linear temporal logic (LTL), a logic designed to express the truth of logical predicates over time. Detailed guides to LTL exist in the literature [26, 29], but a brief overview is as follows:

- $\Box p$, read as "always p", which means, "from now on, it is always the case that p is true".
- $\Box \neg p$, read as "always not p", which means "from now on, it is always the case that p is false".
- $\Diamond p$, read as "eventually p", which means, "at some point, either now or in the future, p will be true".
- $\Diamond(p \wedge q)$, read as "eventually p and q", which means, "at some point, either now or in the future, both p and q will be true".
- $\Diamond(p \vee q)$, read as "eventually p or q", which means, "at some point, either now or in the future, either p or q will be true".
- $p \Longrightarrow q$, read as "p implies q", which means, "if p is true then q is true".
- $p \ \mathsf{U} \ q$, read as "p until q", which means, "from now on, p will be true until q is true, and q must eventually be true".
- $p \ \mathsf{R} \ q$, read as "p releases q", which means, "from now on, q will be true up to the point that p is true".

- $\Box\Diamond p$, read as "infinitely often p", which means, "from now on, p will be always be true eventually".

As rational agents have beliefs, desires and intentions, it is useful to be able to model check properties of programs that involve these constructs. This is done through the use of the B, G, I and A modalities that extend LTL [31], for example:

- $\mathsf{B}_\mathsf{a} p$, read as "agent a believes p".
- $\neg\mathsf{B}_\mathsf{a} p$, read as "agent a does not believe p".
- $\mathsf{G}_\mathsf{a} g$, read as "agent a has a goal g". Goals can be achieve goals or perform-goals (e.g., $!_a\gamma$ and $!_p\gamma$ respectively).
- $\neg\mathsf{G}_\mathsf{a} g$, read as "agent a does not have a goal g".
- $\mathsf{I}_\mathsf{a} i$, read as "agent a intends i".
- $\neg\mathsf{I}_\mathsf{a} i$, read as "agent a does not intend i".
- $\mathsf{A}_\mathsf{a}\alpha$, read as "agent a does α".
- $\neg\mathsf{A}_\mathsf{a}\alpha$, read as "agent a does not have an action α".

These modalities can be combined with LTL operators to form more complex properties, e.g.:

- $\Box\mathsf{B}_\mathsf{a} p$, read as "always agent a believes p", which means"from now on, it is always the case that agent a believes p is true".
- $\Diamond(\mathsf{G}_\mathsf{a}!_p g \implies \mathsf{B} b)$, which means "eventually, if agent a has a perform-goal g then it will have a belief b".
- $\mathsf{G}_\mathsf{a}!_a g \,\mathsf{U}\, \mathsf{I}_\mathsf{a} q$, which means "agent a has an achieve-goal g until agent a intends to do q".

Note that if only one agent is being considered, it is sufficient to say simply "$\mathsf{B}p$", rather than "$\mathsf{B}_\mathsf{a} p$" for some agent "a".

In model checking, *safety properties* describe a state of affairs where something undesirable never happens, e.g., $\Box\neg bad$, where *bad* is some undesirable condition. Other kinds of properties that could be tested include *reachability properties* (for example, $\Diamond good$ — at some point in the future something good will happen), *liveness properties* (for example, $\Box\Diamond good$ — something good keeps happening) or *fairness properties* (for example, $\Box\Diamond send \implies \Box\Diamond receive$ — if a message is sent infinitely often, then it is received infinitely often as well). The properties used for agent model checking in the next section are either safety, reachability or liveness properties.

III. Model Checking Agents for Autonomous Unmanned Aircraft

The Executive agent for controlling the flight of an unmanned aircraft (described in Section II B) was verified using the Agent JPF program model checker. A list of properties that were successfully model checked is below. The properties are divided into two types: those derived from the Rules of the Air [9], and those derived from some concepts of *Airmanship*. The Rules of the Air have been developed over time and are the result of many years of learning. However, this does not imply that they are complete, accurate or even self-consistent. There are many tasks or activities that pilots perform that are not written in the Rules of the Air but that are essential for maintaining safe operations. Indeed, on some occasions, the safest course of action might be to ignore or deliberately break one or more of those rules. To contravene written rules in the name of safety is one possible requirement for "Airmanship." More generally, airmanship requires behaviour which is not necessarily found in the Rules of the Air but would be expected from a competent pilot; for example checking vehicle systems and fuel level before take-off, diverting based on new information about weather, etc. Our airmanship requirements were based on interviews with subject matter experts.

It is worth noting that requirements do not always map onto properties on a one-to-one basis: some properties can cover more than one requirement (for example, properties 1–3 below), and some requirements can require more than one property (for example, property 4 below).

A. Properties based on the Rules of the Air

1. **Detect and Avoid 1**

   Requirement: "When two aircraft are approaching head-on, or approximately so, in the air and there is a danger of collision, each shall alter its course to the right."

   Property:

   $$\Box(\mathsf{B}(\text{exec}, \text{intruderAircraft}) \Longrightarrow \Diamond\mathsf{B}(\text{exec}, \text{flightPhase}(\text{emergencyAvoid})))$$

   Notes: This property says that it is always the case that if the Executive believes that there is an intruder aircraft on a collision course, then it will eventually believe that it has shifted the unmanned aircraft into an "emergency avoid" flight phase. This formalisation is an adequate translation of the requirement as once the `emergencyAvoid` flight phase is selected, the Planner will calculate an emergency avoidance route which will steer the aircraft to the right and around the intruder aircraft.

2. **Detect and Avoid 2**

   Requirement: "Notwithstanding that a flight is being made with air traffic control clearance it shall remain the duty of the commander of an aircraft to take all possible measures to ensure that his aircraft does not collide with any other aircraft."

   Property:

   $$\Box(\mathsf{B}(\text{exec}, \text{intruderAircraft}) \Longrightarrow \Diamond\mathsf{B}(\text{exec}, \text{flightPhase}(\text{emergencyAvoid})))$$

   Notes: The formalisation of this property is the same as that for property 1. This is because to formalise that "all possible measures" are taken by the commander (or, in this case, the Executive) to avoid collisions it is sufficient to state that the rational agent will always change its flight phase to `emergencyAvoid` when it believes there is an intruder aircraft. Based on the design of the Executive and the UAS virtual prototype there is nothing else that the Executive could do in order to avoid a collision, and therefore this formalisation lets us formally verify this requirement.

3. **Detect and Avoid 3**

   Requirement:

   "An aircraft shall not be flown in such proximity to other aircraft as to create a danger of collision."

   Property:

   $$\Box(\mathsf{B}(\text{exec}, \text{intruderAircraft}) \Longrightarrow \Diamond\mathsf{B}(\text{exec}, \text{flightPhase}(\text{emergencyAvoid})))$$

   Notes: The formalisation of this requirement is the same as the formalisation of requirements 1 and 2. This property is an adequate formalisation of this requirement as it is assumed that, whenever there is an intruder aircraft, that it will be detected and the Executive will be notified. Once the Executive is notified, it will always change the flight phase to `emergencyAvoid` (as verified by this property) and this will cause the flight control system to avoid the intruder aircraft based on a plan provided by the Planner. Furthermore it is assumed that the Planner will always choose routes through the environment which will avoid other aircraft, therefore providing an additional layer of protection against the danger of a collision.

4. **ATC Clearance**

   Requirement: "An aircraft shall not taxi on the apron or the manoeuvring area of an aerodrome without the permission of either: (a) the person in charge of the aerodrome; or (b) the air traffic control [ATC] unit or aerodrome flight information service unit notified as being on watch at the aerodrome."

   Property:

   $$\Box(\mathsf{B}(\text{exec}, \text{taxying}) \Longrightarrow \mathsf{B}(\text{exec}, \text{taxiClearanceGiven}))$$

   $$\Box(\mathsf{B}(\text{exec}, \text{lineUp}) \Longrightarrow \mathsf{B}(\text{exec}, \text{lineUpClearanceGiven}))$$

$$\Box(\mathsf{B}(\text{exec}, \text{takeOff}) \Longrightarrow \mathsf{B}(\text{exec}, \text{takeOffClearanceGiven}))$$

Notes: These formalisations adequately describe the requirement as they state that the Executive will not believe that it is doing *a* unless it believes that it has clearance to do *a*. We assume that the Planner (which plans routes around the aerodrome) will only generate routes for which clearance is granted, and that the flight control system (which implements the routes) will not err in its implementation of these routes so that a plan to taxi will not result in the aircraft taking off, for example.

5. **500 feet rule**

   Requirement: "Except with the written permission of the CAA, an aircraft shall not be flown closer than 500 feet to any person, vessel, vehicle or structure."

   Property:

$$\Box\left[\begin{array}{c}\left(\begin{array}{c}\mathsf{B}(\text{exec}, \text{das}(\text{alert500})) \wedge \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{waitingAtRamp})) \wedge \\ \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{taxying})) \wedge \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{taxyingDestination})) \wedge \\ \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{takeOff})) \wedge \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{landing}))\end{array}\right) \\ \Longrightarrow \\ \Diamond\mathsf{B}(\text{exec}, \text{flightPhase}(\text{emergencyAvoid}))\end{array}\right]$$

   Notes: This formalisation of the requirement states that if the Executive believes that the detect-and-avoid sensor (`das`) has alerted that the aircraft is in danger of flying within 500 feet of another airspace user, and if the Executive believes that it is not in one of the ground-based flight phases (i.e., it is flying), then it will eventually change its flight phase to `emergencyAvoid`. Of course, we make the usual assumptions about the DAS, Planner, and flight control systems that they will perform adequately within their own operations so as not to cause the aircraft to violate this Rule.

6. **1000 feet rule**

   Requirement: "Except with the written permission of the CAA, an aircraft flying over a congested area of a city town or settlement shall not fly below a height of 1,000 feet above the highest fixed obstacle within a horizontal radius of 600 metres of the aircraft."

   Property:

$$\Box\left[\begin{array}{c}\left(\begin{array}{c}\mathsf{B}(\text{exec}, \text{das}(\text{alert1000})) \wedge \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{waitingAtRamp})) \wedge \\ \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{taxying})) \wedge \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{taxyingDestination})) \wedge \\ \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{takeOff})) \wedge \neg\mathsf{B}(\text{exec}, \text{flightPhase}(\text{landing}))\end{array}\right) \\ \Longrightarrow \\ \Diamond\mathsf{B}(\text{exec}, \text{flightPhase}(\text{emergencyAvoid}))\end{array}\right]$$

   Notes: The formalisation of this requirement is similar to that of requirement 5, and is based on similar assumptions. The only difference is in the alert type ("alert1000") coming from the detect-and-avoid sensor.

7. **Check-weather-before-flight rule**

   Requirement: "Subject to paragraph (4), an aircraft which is unable to communicate by radio with an air traffic control unit at the aerodrome of destination shall not begin a flight to the aerodrome if: (a) the aerodrome is within a control zone; and (b) the weather reports and forecasts which it is reasonably practicable for the commander of the aircraft to obtain indicate that it will arrive at that aerodrome when the ground visibility is less than 10 km or the cloud ceiling is less than 1,500 feet."

Property:

$$\Box\left[\begin{array}{c}\left(\begin{array}{c}\mathsf{B}(\text{exec},\text{flightPhase}(\text{waitingAtRamp}))\wedge\mathsf{B}(\text{exec},\text{destination}(\text{D}))\wedge\\ \neg\mathsf{B}(\text{exec},\text{location}(\text{D}))\wedge\mathsf{B}(\text{exec},\text{unableToCommunicate}(\text{D}))\wedge\\ (\mathsf{B}(\text{exec},\text{groundVisibilityLow}(\text{D}))\vee\mathsf{B}(\text{exec},\text{cloudCeilingLow}(\text{D})))\wedge\\ \text{controlZone}(\text{D})\end{array}\right)\\ \Longrightarrow\\ \Box\mathsf{B}(\text{exec},\text{flightPhase}(\text{waitingAtRamp}))\end{array}\right]$$

Notes: This formalised requirement states that if the Executive is waiting at the ramp (i.e., at the start of its mission), and has a planned destination *D*, and it believes that it is unable to communicate by radio and either ground visibility is low or the cloud ceiling is low at the planned destination, and it believes that the destination is in a control zone, then the unmanned aircraft will always be in the `waitingAtRamp` flight phase, i.e., it will never taxi, take-off, fly, etc. This adequately captures the requirement if we assume that the sensors which inform the Executive are behaving correctly, and that the flight control system of the unmanned aircraft will not move beyond the `waitingAtRamp` flight phase unless told to by the Executive.

B. Properties based on Airmanship

Airmanship properties are derived from requirements based on interviews with subject matter experts, and represent rules that should be followed by aircraft operating in civil airspace, but which are not explicitly written in the Rules of the Air. These are in no way intended to be exhaustive, but are a representative sample set for the purposes of the work reported in this paper.

The first seven airmanship properties are:

8. **ATC subsystem polling**

   Requirement: "The commander of an aircraft must pay attention to instructions from air traffic control during the operation of the aircraft."

   Property:

$$\Box\Diamond\mathsf{B}(\text{exec},\text{polled}(\text{atc}))$$

   Notes: This formalisation captures the requirement as it says that it is always the case that the Executive (the "commander" of the autonomous unmanned aircraft) will eventually believe that it has polled the air traffic control subsystem for more information. Recall that the Executive polls its subsystems for information periodically. We assume that the air traffic control subsystem will always inform the Executive once polled if it has some new information from air traffic control. As the poll beliefs are deleted periodically, this formalisation allows verification of this requirement.

9. **DAS subsystem polling**

   Requirement: "The commander of an aircraft must pay attention to potential intruder aircraft during flight."

   Property:

$$\Box\left(\begin{array}{c}(\neg\mathsf{B}(\text{exec},\text{flightPhase}(\text{cruise}))\wedge\neg\mathsf{B}(\text{exec},\text{flightPhase}(\text{emergencyAvoid})))\vee\\ \Box\Diamond\mathsf{B}(\text{exec},\text{polled}(\text{das}))\end{array}\right)$$

   Notes: This formalisation says that the Executive will always eventually believe that it has polled the detect and avoid sensor if it is in the `cruise` or `emergencyAvoid` flight phases. This is because during other flight phases the detect-and-avoid sensor is not polled as it is only use during the airborne (i.e., `cruise` and `emergencyAvoid`) flight phases. The usual assumptions concerning sensor accuracy, planner reliability and flight control system accuracy apply.

10. **Env subsystem polling**

    Requirement: "During the operation of an aircraft the commander must pay attention to changing environmental situations, including inclement weather, aerodrome and airspace closures, and so on."

    Property:

    $$\Box\Diamond \mathsf{B}(\text{exec}, \text{polled}(\text{env}))$$

    Notes: The justification for this formalisation is similar to that for property 8, but with the ATC subsystem replaced with the Env subsystem.

11. **Fuel subsystem polling**

    Requirement: "The commander of an aircraft must pay attention to fuel levels during operation of the aircraft."

    Property:

    $$\Box\Diamond \mathsf{B}(\text{exec}, \text{polled}(\text{fuel}))$$

    Notes: The justification for this formalisation is similar to that for property 8, but with the ATC subsystem replaced with the Fuel subsystem.

12. **Nav subsystem polling**

    Requirement: "During the operation of an aircraft, the commander must pay attention to the aircraft's location."

    Property:

    $$\Box\Diamond \mathsf{B}(\text{exec}, \text{polled}(\text{nav}))$$

    Notes: The justification for this formalisation is similar to that for property 8, but with the ATC subsystem replaced with the Nav subsystem.

13. **Planner subsystem polling**

    Requirement: "During the operation of an aircraft, the commander must regularly check the planned navigation route in light of new information, and modify the planned navigation route based on new information if necessary."

    Property:

    $$\Box\Diamond \mathsf{B}(\text{exec}, \text{polled}(\text{planner}))$$

    Notes: The justification for this formalisation is similar to that for requirement 8, but with the ATC subsystem replaced with the Planner subsystem. In addition, we assume that the Planner is given a destination by the Executive at the start of the mission, and responds by giving the Executive a route to follow. During the mission the Planner is continually assessing the route given to the Executive in light of changes in the environment, such as weather or other aircraft (for example), and will re-plan a route should the need arise. If the Planner determines that a new route is required and has generated such a route, then we assume that it will notify the Executive the next time it is polled. Therefore this formalisation adequately describes this requirement.

14. **Veh subsystem polling**

    Requirement: "During the operation of an aircraft, the commander must regularly check that the aircraft is in good working order."

    Property:

    $$\Box\Diamond \mathsf{B}(\text{exec}, \text{polled}(\text{veh}))$$

    Notes: The justification for this formalisation is similar to that for property 8, but with the ATC subsystem replaced with the Veh subsystem.

15. **Complete mission**

    Requirement: "During the operation of an aircraft, the commander of an aircraft should endeavour to complete the stated mission until that mission is complete."

    Property:

$$\Diamond(\mathsf{B}(\text{exec}, \text{missionComplete}) \quad \mathsf{R} \quad \mathsf{G}(\text{exec}, !_{\text{a}}\text{completeMission}))$$

Notes: This requirement concerns the Executive agent's behaviour towards the completion of its mission. The formalisation of the requirement states that eventually the Executive must have a continuous goal to complete its mission that is *released* by the belief that its mission is complete. This is a relatively strict requirement for the Executive, as it does not allow for the Executive to delete its goal to complete the mission after deciding that the goal is impossible to achieve. It may be preferable to allow the Executive to "give up" in trying to complete its mission, but in this specific case, we do not permit this.

16. **No further activity after mission is complete**

    Requirement: "Once the mission is complete, the commander of an aircraft does not need to try to complete the mission any longer."

    Property:

$$\Box\left(\begin{array}{c}\mathsf{B}(\text{exec}, \text{missionComplete}) \quad \Longrightarrow \\ \Diamond\Box(\neg\mathsf{G}(\text{exec}, !_{\text{a}}\text{completeMission}) \wedge \neg\mathsf{I}(\text{exec}, \text{completeMission}))\end{array}\right)$$

    Notes: This requirement sounds very abstract for a human commander of an aircraft; it is almost too obvious to state that once a person has achieved what they set out to do that they no longer need to try to achieve it. However, in the case of an autonomous system these kinds of facts cannot be considered obvious, and must be listed as system requirements.

    The formalisation of this requirement states that once the Executive believes that the mission is complete, then it will eventually be the case that the executive will never have a goal to complete the mission, and will never have an intention to complete the mission. This, in effect, states that once the Executive believes that its mission is complete then it will eventually stop trying to complete the mission. Of course, this assumes that the Executive will only ever have one mission per execution run. If it were desirable for the Executive to have multiple missions, then this requirement would need to be modified. (Another alternative to multiple missions for the Executive would be to have a single "overriding" mission which is to complete a set of sub-missions, in which case this requirement could remain.)

17. **Check fuel before taxi**

    Requirement: "The commander of an aircraft should check the current fuel level before starting to taxi."

    Property:

$$\Box(\mathsf{A}(\text{exec}, \text{send}(\text{planner}, \text{enactRoute}, \text{taxi}, \text{Num})) \Longrightarrow \mathsf{B}(\text{exec}, \text{fuel}(\text{taxi}, \text{level}, \text{M}))$$

    Notes: This requirement is derived from requirement 11: "The commander of an aircraft must pay attention to fuel levels during operation of the aircraft." However it is of sufficient importance to the safe operation of the unmanned aircraft that we may wish to verify this requirement as a special case. The formalisation states that, "It is always the case that when the Executive sends a message to the planner to enact a taxi route that it has a belief concerning the current fuel level." This formalisation adequately represents the requirement as long as we assume that the fuel sensor systems are working properly (e.g., not reporting old fuel levels rather than the current one).

18. **Check fuel before cruise**

    Requirement: "The commander of an aircraft should check the current fuel level before starting to cruise."

Property:

$$\Box(\mathsf{A}(\text{exec}, \text{send}(\text{planner}, \text{enactRoute}, \text{cruise}, \text{Num})) \Longrightarrow \mathsf{B}(\text{exec}, \text{fuel}(\text{cruise}, \text{level}, \text{M}))$$

Notes: This requirement is similar to requirement 17 and is also a special case of requirement 11. The formalisation adequately describes the requirement under the same assumption as requirement 17.

19. **Check fuel before approach**

Requirement: "The commander of an aircraft should check the current fuel level before starting the approach to land."

Property:

$$\Box(\mathsf{A}(\text{exec}, \text{send}(\text{planner}, \text{enactRoute}, \text{approach}, \text{Num})) \Longrightarrow \mathsf{B}(\text{exec}, \text{fuel}(\text{approach}, \text{level}, \text{M}))$$

Notes: This requirement is also similar to requirement 17 and is also a special case of requirement 11. The formalisation adequately describes the requirement under the same assumption as requirement 17.

20. **Divert based on Flight Information Service**

Requirement: "The commander of an aircraft must re-plan any route that is affected by a notice of airspace closure by a Flight Information Service."

Property:

$$\Box(\mathsf{B}(\text{exec}, \text{env}(\text{fis}, \text{Desc}, \text{Fir}, \text{Lat}, \text{Lon}, \text{Radius}, \text{Height})) \Longrightarrow \Diamond\mathsf{B}(\text{exec}, \text{enactRoute}(\text{divert}, \text{Num})))$$

Notes: This formalisation states that if the Executive believes that it has a message from a Flight Information Service then it will eventually believe that it has enacted a route to divert based on that message. This requirement concerns what happens when the Executive receives information during flight from a Flight Information Service (FIS): a service provided by a national aviation administration (such as the FAA or CAA) that supplies advice and information to aircraft to enable safe and efficient flight. FISs can provide information on weather, conditions at aerodromes or any other information likely to affect safety. (For a description of FISs see [32].) Typically a FIS will broadcast information via radio to alert pilots of new information that may affect aircraft flying in a given Flight Information Region (FIR). For instance, an FIS may send an alert that there is a smoke cloud at latitude 53.3° north and longitude 2.6° west in the London flight information region affecting airspace at a radius of 5 nautical miles and up to 10,000 feet in height. In the case of our autonomous unmanned aircraft, the Executive agent would receive a message from the Env subsystem that airspace is affected for the current planned route. On receiving this message the Executive will enact a new route by sending a message to the Planner. This behaviour was verified using the formalisation above, which replaces specific details like "smoke cloud", "UK" and "53.3° north" with variables "Desc", "Fir", "Lat", etc. (Variables always begin with upper case letters in the Agent JPF property specification language.)

This formalisation adequately describes the requirement as long as we assume that the Env and Planner subsystems are operating correctly and are not reporting false FIS reports (in the case of Env) or returning invalid plans (in the case of the Planner).

21. **Divert based on Flight Information Service (stronger version)**

Requirement: "The commander of an aircraft must re-plan any route that is affected by a notice of airspace closure by a Flight Information Service."

Property:

$$\Box\left[\begin{array}{l}\mathsf{B}(\text{exec}, \text{env}(\text{fis}, \text{Desc}, \text{Fir}, \text{Lat}, \text{Lon}, \text{Radius}, \text{Height})) \\ \Longrightarrow \\ \Diamond\left(\begin{array}{l}\mathsf{A}(\text{exec}, \text{send}(\text{planner}, \text{reqRoute}, \text{fis}, \text{Desc}, \text{Fir}, \text{Lat}, \text{Lon}, \text{Radius}, \text{Height}, \text{L})) \\ \wedge \\ \Diamond\left(\begin{array}{l}\mathsf{B}(\text{exec}, \text{route}(\text{divert}, \text{Num}, \text{Time}, \text{Fuel}, \text{Safety})) \\ \wedge \\ \Diamond\mathsf{A}(\text{exec}, \text{send}(\text{planner}, \text{enactRoute}, \text{divert}, \text{Num}))\end{array}\right)\end{array}\right)\end{array}\right]$$

Notes: This requirement is a stronger version of requirement 20. This formalisation states that it is always the case that, if the Executive believes that it has received a message from a Flight Information Service, then it will eventually send a request to the Planner subsystem for a diversion route based on the new information, and it will eventually receive a plan back from the Planner and will then eventually send a message to the Planner to enact that plan. This formalisation adequately describes the requirement given the same assumptions concerning the Env and Planner subsystems.

IV. Higher Fidelity Environment Models for Model Checking

Webster et al. [8] describe how rational agents for unmanned aircraft can be model checked using a model of the agent's environment consisting of three components:

- A *sensor unit*. This represents a detect-and-avoid sensor which alerts the agent whenever an object is approaching head-on.
- A *navigation manager*. This represents a navigation subsystem which alerts the agent whenever the vehicle must correct its course in order to reach a pre-planned destination.
- An *air traffic control transceiver*. This communicates with aerodrome air traffic control and notifies the agent if different clearances have been given or denied, for example clearance to taxi or take-off.

(This is an earlier prototype of the system described in Section II B.) For example, the sensor unit may send a message, "aircraftApproachingHeadOn" to alert the agent that this is the case, and may send a message "aircraftAverted" to alert the agent that the aircraft is no longer approaching head on. The navigation manager and air traffic control models behave in a similar way. It is clear that the model of the environment is explicit relative to the messages being sent between components, but *implicit* relative to physical reality. For example, the position of the intruder aircraft is not modelled explicitly; nowhere in the model is this data stored. Rather, just the *effect* of an intruder aircraft is modelled; in this case, an alert from a sensor unit saying that there is an intruder aircraft. Likewise, the navigation of the vehicle through airspace is not modelled explicitly, nor are the air traffic conditions at the aerodrome under air traffic control.

This kind of environment modelling is based on an understanding of the design of the autonomous system in which the agent is based. All that is needed is an understanding of the protocol by which the agent communicates with other (electronic) components on the unmanned aircraft and how those components reflect and report the physical world to the agent. Indeed, this kind of protocol-based model has been used with considerable success when model checking mission-critical software, (see, for example, [33]).

However, in the case of agent model checking for unmanned aircraft, this kind of implicit environment modelling presents difficulties. Frequently it is desirable to verify the behaviour of an agent-based autonomous unmanned aircraft using the Rules of the Air, many of which include physical quantities, e.g., the "500 feet" and "Check-weather-before-flight" rules in Section III A.

In implicit models, these physical quantities are not present by definition. For example, a sensor system may report that an intruder aircraft is "approaching", but that aircraft's physical attributes such as heading, speed, altitude, etc., are not modelled explicitly. In order to formally verify properties which contain physical quantities, a more explicit environment model is required. The use of more realistic environment models within model checking has a number of advantages over simulation and other model checking techniques such as those described by Webster et al. [8]:

1. Physical quantities can be modelled, meaning that requirements based on physical quantities (such as those given above) can be verified formally.
2. Flight simulation software can be used to verify the assumptions made in the environment model. In this case, simulation is not being used directly to generate evidence for certification but is helping to verify the assumptions made in the model being used for certification.

A. Developing a Higher Fidelity Environment Model for Model Checking

In order to develop a higher fidelity environment model for model checking, a detect-and-avoid scenario that had previously been developed in a flight simulation was analysed. In the scenario, an unmanned aircraft is flying straight and level towards a route waypoint through simulated UK civil airspace in the "cruise" flight phase, and encounters a single intruder aircraft approaching (approximately) head-on. A sensor system

registers the intruder, informing the Executive agent, which decides to implement an "emergency avoid" manoeuvre consisting of a turn to the right (in accordance with the Rules of the Air). When the sensor system registers that the intruder aircraft has been successfully avoided, it informs the Executive agent which decides to cancel the Emergency Avoid manoeuvre and return to the "cruise" flight phase. When the flight control system receives this command, it executes a turn to the left in order to resume its path towards the next waypoint on the route. The manoeuvre is illustrated in Fig. 5.

(Here we are examining a scenario in which an intruder aircraft is approaching head-on, or approximately so. It is possible to examine other scenarios in which the intruder aircraft is approaching from different directions, but for the purposes of this example we choose an intuitive "head on" scenario. In fact, the Executive is flexible with regards to the direction of approach of the intruder aircraft, as its response is the same in each case: to change the flight phase to `emergencyAvoid`, causing the Planner to calculate a route around the intruder aircraft and the flight control system to implement that route.)

(a) (b) (c)

Fig. 5 A simulated detect-and-avoid manoeuvre in the VESL flight simulator. Image (a) is prior to the manoeuvre, image (b) is after the first turn and image (c) is after the second turn. These turns correspond to the turns shown later in Figures 7 and 8.

The flight simulation environment used is the VEC's Virtual Engineering Simulation Laboratory (VESL) described by Cameron et al. [10]. VESL uses Advanced Rotorcraft Technology's FLIGHTLAB software [34] for high fidelity flight dynamics simulation. FLIGHTLAB is connected via custom software and hardware interfaces to the Executive agent described in Section II B. The agent is kept informed of changes in its environment through simulated sensors, one of which is a detect-and-avoid sensor system. Once the Executive agent has decided on a course of action, it sends a message to the flight control system (also simulated in FLIGHTLAB) which determines the control inputs required to execute the Executive agent's command. For example, the Executive agent may decide to alter course to a heading of 210°, for which it will send a message to the flight control system. The flight control system will then calculate and implement the exact combination of simulated actuator inputs required to cause the vehicle to turn onto the desired heading. A schematic illustration of the VESL's architecture is given in Fig. 6.

As the aim was to create a higher fidelity environment model for the purposes of model-checking the Executive agent, the detect-and-avoid manoeuvre from the flight simulation was recorded and analysed. Since the manoeuvre consists of two turns, one to the right and one to the left, the headings of the unmanned aircraft in the simulation were noted at three points: (i) prior to the manoeuvre, (ii) after the first turn; and (iii) after the second turn. Furthermore, the duration of time between (ii) and (iii) was noted. The sensor model used in the detect-and-avoid simulation was a simple proximity sensor modelled within the flight dynamics modelling system (FLIGHTLAB), set to detect any intruder aircraft within 20,000 feet (6,096 metres).

(Note that this proximity sensor was a simple "placeholder" model designed to work in the case of a single uncooperative intruder aircraft, and did not monitor whether the intruder was on a collision course. Of course, the proximity sensor model works in the scenario being examined here. Higher fidelity detect-and-avoid sensor models are being developed to tackle more elaborate scenarios.)

This proximity sensor was represented within the model checking environment model as a finite state machine written in Java. The intruder aircraft in the simulation was set to approach the unmanned aircraft head-on, and this was modelled in the model checking environment using a Java-based representation of the simulated airspace.

The Executive agent was reduced to a core set of plans necessary for detect-and-avoid functionality. The Executive agent's execution environment was adapted to provide a simple (yet explicit) physical model of the unmanned aircraft's flight through airspace, based on heading, speed, latitude and longitude. (The altitude was assumed to be constant for the unmanned aircraft and the intruder, as the altitudes of the unmanned aircraft and the intruder were constant for the duration of the scenario in the VESL flight simulator.)

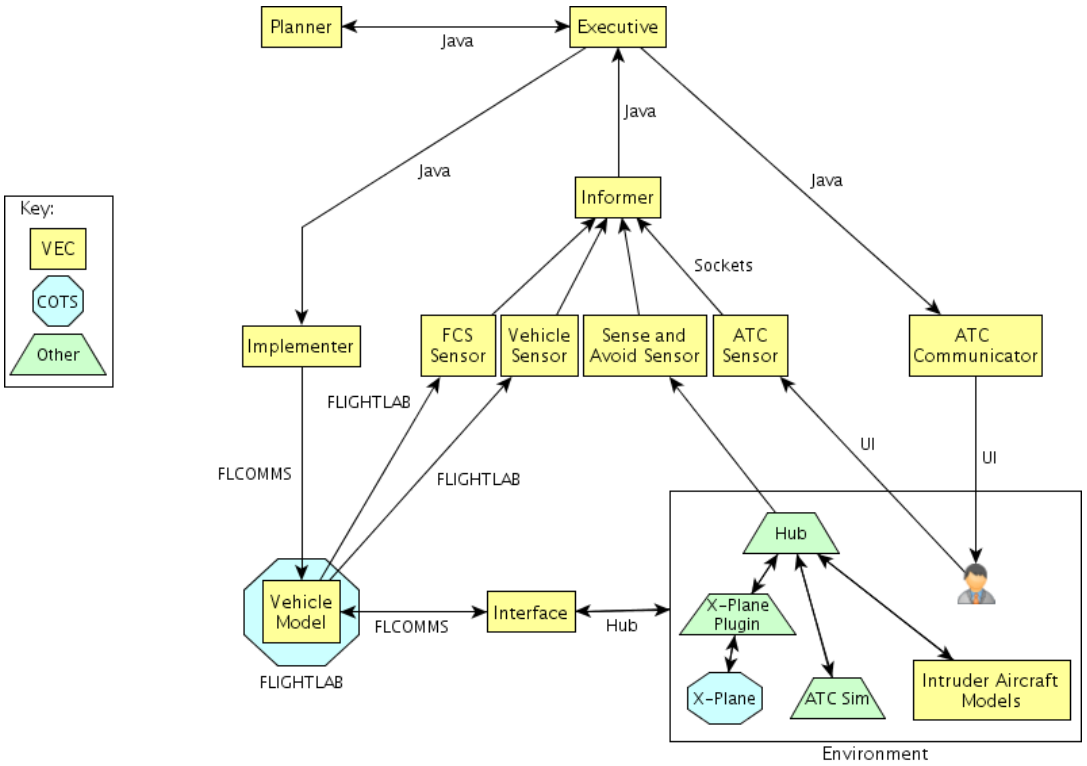


Fig. 6 Software architecture of the Virtual Engineering Simulation Laboratory (VESL) at the VEC. The differently-shaped boxes represent the different origins of the software systems. Arrows represent information flow between software systems, and labels on the arrows show the software/protocol used. More information on the VESL is provided in [10].

When the Executive agent is executed within the higher fidelity environment model, the agent directs the modelled aircraft through the simulated airspace. The intruder aircraft flies in the opposite direction to the unmanned aircraft on a head-on collision course. At a distance of 20,000 feet, a sensor system registers the presence of the intruder aircraft and informs the Executive agent. The Executive then decides to engage the "emergency avoid" flight phase. The manoeuvre described above is conducted until the sensor system detects that the intruder aircraft has gone out of range and informs the Executive agent. At this point the Executive resumes the "cruise" flight phase. Illustrations of the detect-and-avoid scenario as modelled in the higher fidelity environment model are given in Fig. 7.

B. Model Checking Using the Higher Fidelity Environment Model

Using the higher fidelity environment model it was possible to formally verify a Rule of the Air based on physical quantities used in the model. The Executive agent was model checked with respect to requirement 5, but this time with a new formalisation:

22. **The 500 feet rule**:

    Requirement: "... an aircraft shall not be flown closer than 500 feet to any person, vessel, vehicle or structure."

    Property:

$$\Box\neg \mathsf{B}(\mathrm{exec}, \mathrm{separationUnsafe})$$

Here "separationUnsafe" is a statement that is true if, and only if, the distance between the unmanned aircraft and the intruder aircraft is less than 500 feet. The belief "separationUnsafe" will be added to the Executive's belief database only in the event that the distance is less than 500 feet. Therefore if this property holds (i.e., is verified as true by the model checker) then we know that the Executive agent follows this Rule of the Air.

In this case, the model checker checks only the case where the sensor can detect aircraft at a range of up to 20,000 feet, i.e., it checks a single run of the model in which the unmanned aircraft performs the detect-and-avoid manoeuvre. Note that model checking the program in this case is different from simply executing the program once. In execution, the Java runtime environment executes the Java bytecode, performing whichever outputs are specified in the program. However, in model checking, the Agent JPF model checker executes the Java bytecode instead, and examines every step in the execution path to check whether the property being checked is true. If at any point the property evaluates to false, the model checker will stop execution of the program, along with an output about the error-causing state and how that state came to be (i.e., an error trace detailing the states leading up to the error state). It is possible to arrive at the conclusion that model checking in this case is equivalent to testing; however the fact that the state is being analysed at every point together with error trace data according to properties *specified formally* means that model checking offers more than software testing, even in this simple case.

It is possible to use the model checker in a more complex way to analyse a range of different scenarios automatically, and therefore give a high level of assurance that the property being checked holds in all relevant cases. The example above was adapted to vary the sensor detection range r between 16,700 feet and 20,000 feet in 3.28 feet (= 1 metre) intervals in order to determine the effects of reduced sensor range on the detect-and-avoid manoeuvre.

(The reason 16,700 feet was chosen is that this is approximately 1000 metres less than the maximum sensor range of 20,000 feet. The use of two different units for distance was an unavoidable result of the use of more than one simulation environment in this work.)

The variable r was built into the environment model using the Verify.getInt(x,y) method in the Agent JPF model checker, which allows a variable to be altered across a range during model checking run-time. (In the case of execution outside the model checker, i.e., in the Java runtime environment, this method returns a pseudorandom number in the specified range, meaning that during normal "simulation-style" execution the sensor detection range will vary randomly.) The relevant line of Java was:

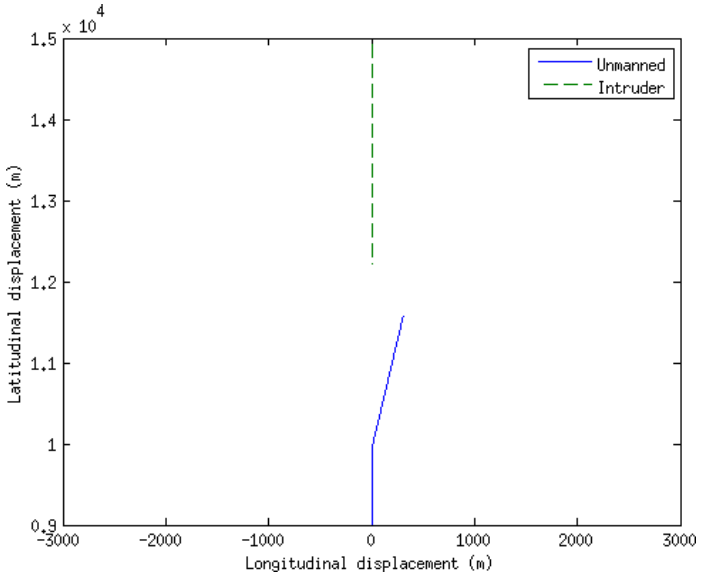
```
sensorRange = (double) Verify.getInt(5096, 6096);        // 6,096m = 20,000 feet
```

The model checker was used to verify formally that the property above was satisfied for all sensor ranges from 16,700 feet to 20,000 feet in 3.28 feet (i.e., 1 metre) intervals.

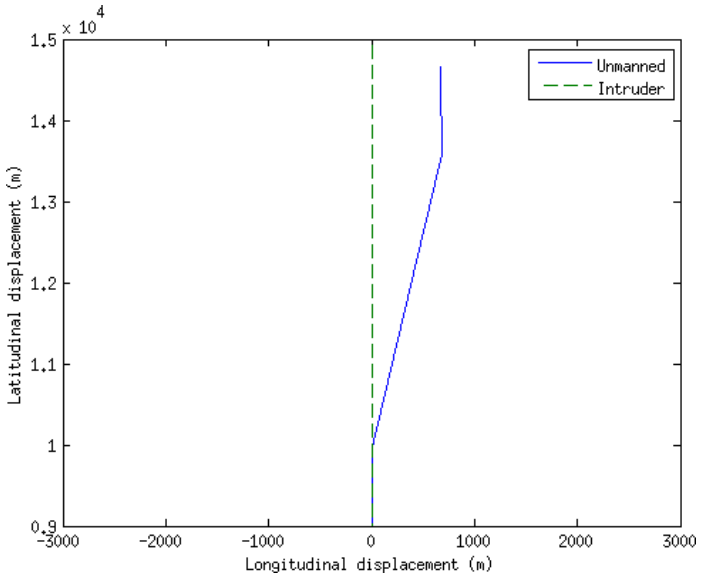
C. Verifying Model Checking Results using Simulation

After execution of the program containing the Executive agent and the higher fidelity environment model, it was found that the minimum separation distance from the unmanned aircraft to the intruder aircraft was 1,129 feet when the detect-and-avoid sensor range was set to 20,000 feet. Analysis of the flight simulation of the same scenario showed that the minimum separation distance between the unmanned aircraft and the intruder aircraft was 820 feet. Clearly there is a difference between the two models of 309 feet. After comparing the outputs of the two models, it was found that in the higher fidelity environment model, the turns in the Emergency Avoid manoeuvre were based only on changes in the heading of the modelled vehicle. For example, the vehicle turns immediately from a heading of 0° to 10.6° in a discontinuous way (see Fig. 7). It was hypothesised that the reason for the 309 feet disparity was due, in part, to the relatively unrealistic way in which the turns were modelled compared to the flight simulation, in which the turns were gradual and continuous. So, in order to improve the accuracy of the higher fidelity environment model, the output of the simulator model was analysed to discover the turn rate of the unmanned aircraft during its two turns. It was found that the turn rate in both turns was 0.83° per second. The higher fidelity environment model was then modified to incorporate this turn rate. Running the simulation again revealed that the minimum separation distance of the unmanned aircraft and the intruder had now reduced to 929 feet — a 109 feet difference, a significant improvement over the original 309 feet difference. The effect of the revised higher fidelity environment model on the unmanned aircraft's flight path can be seen in Fig. 8. (Note that it would be possible to remedy the 109 feet difference through further analysis of the differences between the higher fidelity environment model and the simulator model. For example, one possible cause could be the use of a roll rate in the simulator model, which was not used in the higher fidelity environment model due to project constraints.)

After improving the accuracy of the environment model, the Executive agent was model checked again to ensure that the 500 feet rule requirement was still satisfied for all sensor ranges between 16,700 feet and 20,000 feet.



(a)

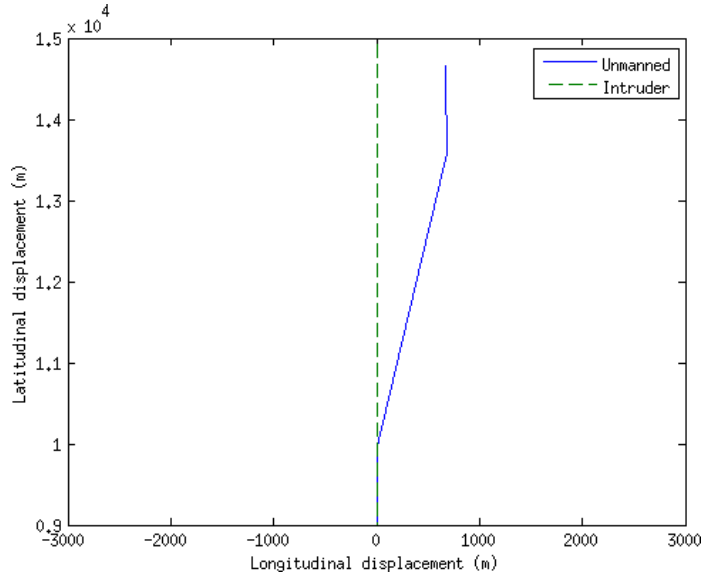


(b)

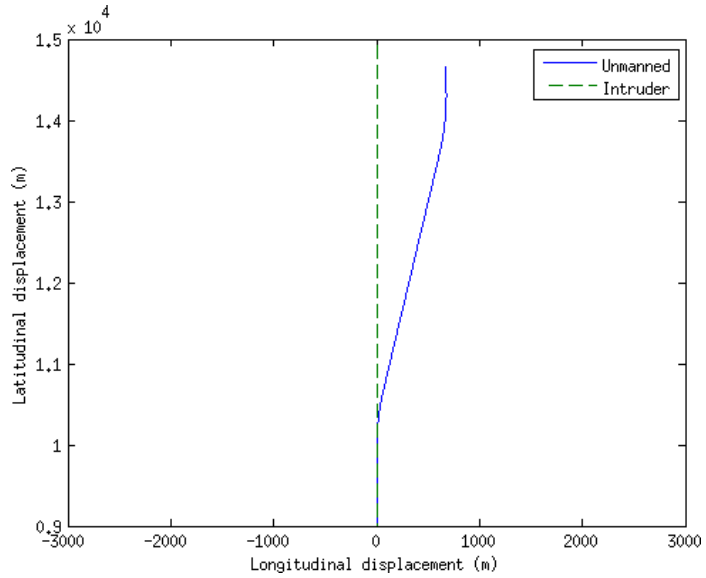
Fig. 7 A simulated detect-and-avoid manoeuvre, as captured by the higher fidelity environment model. Diagrams (a) and (b) show the manoeuvre at the same points as in Figs. 5(b) and 5(c) respectively. The unmanned aircraft is shown as a solid line and starts its run at the bottom of the figure, and the intruder aircraft is shown as a dashed line and starts from the top of the figure. Note that the intruder aircraft is uncooperative and does not alter its course in response to the presence of the unmanned aircraft.

D. Environment Model Fidelity Trade-Offs

In the examples above we contrast an implicit environment model, in which physical quantities are modelled only implicitly, with explicit environment models in which physical quantities are present. In fact, this dichotomy is an over-simplification as there are a wide range of possible environment models across different levels of "explicitness", or fidelity. For example, we could have an environment model in which an aircraft's position is modelled in terms of a sensor alert, or a sensor alert based on heading information, or a sensor alert based on heading and speed information, and so on. Even when we run out of new kinds of information, the accuracy of that information could vary, so that we have a sensor alert that is accurate to one decimal place, or two decimal places, and so on. This leads to an infinite number of different environment models for a simple detect-and-avoid unmanned aircraft scenario.



(a)



(b)

Fig. 8 Differences between the higher fidelity environment model and the revised higher fidelity environment model after including a turn rate in the detect-and-avoid manoeuvre. Diagram (a) shows the manoeuvre in the higher fidelity environment model, and diagram (b) shows the same manoeuvre in the revised higher fidelity environment model. Note that in diagram (b) the turns are smoother than in diagram (a).

One might suspect that as the fidelity of the environment model used for model checking increases that the results obtained from the model increase in reliability. However, this is not necessarily the case. For example, suppose we have an environment with just a single detect-and-avoid sensor which provides one bit of information: whether there is an intruder aircraft approaching head-on, or not. In this case, we can cover all possibilities for the Executive's execution by using the model checker to check both states of the bit: true and false. Adding a higher fidelity environment model in which the speeds and positions of different aircraft are modelled more explicitly may produce more interesting simulations when the model is executed, but will not produce more states of the Executive than are model checked by simply exploring the whole decision space using the model checker by varying the bit representing the detect-and-avoid sensor. In fact, using a higher fidelity environment model may reduce the number of states checked during model checking, and thus result in a loss of generality. For example, in a higher fidelity model, an intruder aircraft may never approach

head-on, thus never triggering the sensor in any state, and therefore causing this branch of the Executive's execution to never be analysed.

However, as we have seen in the examples above, there are some properties that we may wish to model check which are based on numeric quantities. In this case, we must use numeric quantities in the environment model, i.e., higher fidelity environment models, in order to enable the property to be model checked. As we have just described, this may result in a loss of generality of the model. This presents us with a trade-off based on the fidelity of the environment model used: if we use lower fidelity models in which physical quantities are modelled only implicitly, then we are limited in the kinds of property that we can verify using model checking. Specifically, we are limited to those properties which do not involve physical quantities. However, by increasing the fidelity of the model to allow for more properties, we may suffer a loss of generality of the model. It is possible to mitigate against this trade-off by using multiple environment models across varying levels of fidelity, from models in which physical quantities are purely abstract, to those in which they are modelled explicitly as real numbers.

Furthermore, we can design the environment models themselves to take full advantage of the automated nature of model checking. In the example above, we vary the sensor range from 5000 metres to 6000 metres using 1 metre increments. However, we might also vary other aspects within the model, such as the initial point of the intruder aircraft, or its speed, or its heading and so on. This will inevitably result in an increased amount of time required to verify the program using a model checker. More states will need to be checked, but this will result in a higher level of assurance that the program being model checked meets its requirements.

V. Virtual Engineering Using Model Checking

Formal model checking is not limited to proving properties once a system has been implemented; it is frequently a useful tool for guiding the development of the system.

In order to illustrate the way in which model checking can be used for virtual engineering, an example is given detailing how an error found using the model checker was corrected. After testing the virtual prototype VEC123 within a detect-and-avoid scenario using the VESL flight simulator, the Executive agent was model checked using the properties from Section III. Most were found to hold, but some did not. In the case where they did not, the model checker reported that the property was violated. One such error message was as follows:

```
Report for examples.gwendolen.uav.inputsensors16.UavAgent
====================================================== results
error #1: mcapl.MCAPLListener "MCAPL Error is:
AccRun: AccState:true AllSat:fals..."
====================================================== statistics
elapsed time:       0:05:41
states:             new=28, visited=1, backtracked=0, end=0
search:             maxDepth=27, constraints=0
choice generators:  thread=2, data=27
heap:               gc=14161, new=7200081, free=7187869
instructions:       563626936
max memory:         107MB
loaded code:        classes=281, methods=3800
======================================================
search finished: 9/12/12 4:18 PM
```

The error indicated that the following property did not actually hold in all possible scenarios:

23. **Fuel levels are always deleted eventually**:

    Requirement: "Once the Executive has a belief about a fuel level, it should eventually delete that belief."

    Property:

$$\Box(\mathsf{B}(\text{exec}, \text{fuel}(\text{level}, \text{L}, \text{N}) \Longrightarrow \Diamond\neg\mathsf{B}(\text{exec}, \text{fuel}(\text{level}, \text{L}, \text{N})))$$

The property states that it is always the case that if the Executive believes it has a certain fuel level, then eventually it will not believe it has that fuel level. This means that the Executive eventually deletes any belief

it has about a fuel level. This is a way to determine that the Executive is handling fuel beliefs properly: if the above property were not true, it would mean that the Executive had not deleted one of its fuel beliefs. As a result it may be the case that the Executive has more than one belief concerning the current fuel level; not deleting a fuel level belief would result in the Executive having two fuel level beliefs the next time the Executive requests its fuel level from the Fuel subsystem. This could be hazardous as the Executive may believe it has two or more different fuel levels — only one of which would be correct. This could cause the Executive to approve a route for which it has insufficient fuel.

In order to find the source of the error, the Executive was model checked for the same property in a "verbose mode". This allows detailed information on the agent's state to be viewed at every stage during the agent's simulation. Verbose mode is normally turned off as it increases the amount of time required to model check the agent; however, it is often useful for debugging an agent, as in this example. Using verbose mode the state which caused the error was found:

```
AGENT: exec                                                                    1
After Stage StageC :                                                           2
BELIEFS:                                                                       3
 {direction/1=direction(none)[source(self)],                                   4
 enactTaxi/1=enactTaxi(0)[source(self)],                                       5
 position/3=position(52,0,1)[source(self)],                                    6
 enactRoute/2=enactRoute(cruise,0)[source(self)],                              7
 enactRoute(taxi,0)[source(self)],                                             8
 route/5=route(cruise,0,200,100,80)[source(self)],                             9
 route(taxi,0,5,1,90)[source(self)],                                           10
 location/1=location(sumburgh)[source(self)],                                  11
 enactAppr/1=enactAppr(0)[source(self)],                                       12
 destination/1=destination(sumburgh)[source(self)],                            13
 fuel/3=fuel(level,200,2)[source(self)],                                       14
 flightPhase/1=flightPhase(waitingAtRamp)[source(self)],                       15
 connectionOk/0=connectionOk[source(self)], }                                  16
GOALS:                                                                         17
_pstartup(""), _amissionComplete(""),                                          18
CURRENT INTENTION:                                                             19
   *  +polled(self)||(True)||poll(veh)("")||{}                                 20
   *  +polled(self)||(True)||lock("")||{}                                      21
source(self)                                                                   22
OTHER INTENTIONS:                                                              23
[   *     *  x!_amissionComplete("")||(True)||npy("")||{}                       24
   *  +!_amissionComplete("")||(True)||npy("")||{}                             25
   *  +!_pstartup("")||(True)||+!_amissionComplete("")("")||{}                 26
   *  start||(True)||null("")||{}                                              27
source(self),    *  xpolled(self)||(True)||npy("")||{}                        28
self]                                                                          29
```

Line 14 shows the fuel belief, fuel(level, 200, 2). This fuel level is a representation of a more accurate flight simulation-based fuel level, and indicates that the fuel level is 200 litres and that this is the second fuel level message received by the Executive. Line 15 shows the flight phase at the point of the error, "waitingAtRamp". This, together with the belief *location*(*sumburgh*) in line 11, indicates that the error occurred at the end of the Executive mission after the aircraft had landed at Sumburgh airport. This information indicated that the error could have been caused by the Executive's plans concerning approach and landing, as this is the last point during the modelled flight at which the Executive checks its fuel level. The plans concerning approach and landing were as follows:

+*veh*(*landed*,*L*) : {*B destination* (*L*), *B flightPhase* (*approach*)} <− lock, −*veh*(*landed*,*L*), 1
−*flightPhase*(*approach*), +*flightPhase* (*landed*), *send*(*veh*,*updatefp*,*landed*), *send*(*fuel* , *reqfuelstatus*),
+*waiting*(*fuel* ,*landed*), unlock;

+*fuel*(*level* ,*F*,*N*) : {*B waiting*(*fuel* ,*landed*)} <− lock, −*waiting*(*fuel* ,*landed*), *send*(*nav*, *reqposition*), 2
+*waiting*(*nav*,*landed*), unlock;

+*position* (*Lat*,*Lon*,*Alt*) : {*B destination* (*D*), *B waiting*(*nav*,*landed*), *B fuel*(*level* ,*F*,*N*)} <− lock, 3
−*waiting*(*nav*,*landed*), *send*(*planner*, *reqtaxiDestination* ,*D*,*Lat*,*Lon*,*F*), +*waiting*(*planner*,*landed*),
−*fuel*(*level* ,*F*,*N*), unlock;

+*taxi*(*Num*,*Time*,*Fuel*,*Safety*) : {*B waiting*(*planner*,*landed*)} <− lock, −*waiting*(*planner*,*landed*), 4
send(*planner*, *enactTaxi* ,*Num*), +*waiting*(*planner*, *enactTaxi*), unlock;

```
+enactTaxi(Num) : {B waiting(planner, enactTaxi), B flightPhase(landed)} <- lock,                    5
    -waiting(planner, enactTaxi), -flightPhase(landed), +flightPhase( taxyingDestination ),
    send(veh, updatefp, taxyingDestination ), -taxi(Num,Time,Fuel,Safety), +polled(veh), unlock;
+poll( self ) : {B veh( status ,waitingAtRamp), B destination (L), B location (L), B                 6
    flightPhase ( taxyingDestination )} <- lock, -veh(status ,waitingAtRamp),
    -flightPhase( taxyingDestination ), +flightPhase (waitingAtRamp), +polled( self ), unlock;
```

It can be seen that in line 1 the Executive agent requests its current fuel status from the Fuel subsystem: send(fuel,reqfuelstatus). The Fuel subsystem then sends the current fuel status to the Executive agent, which then adds a belief concerning the current fuel level. In line 3 this belief is deleted: −fuel(level,F,N). Therefore the error did not arise from a malfunction in this part of the agent.

After further examination of the Executive agent's Gwendolen code, it was determined that there were three different points during the Executive agent's execution when it would receive a fuel level update: during planning to taxi at the start of the mission, during planning to approach the aircraft, and during planning to taxi to the ramp at the end of the mission. However there were only two corresponding points in the Executive agent's execution where it deleted the belief about its fuel level, thus causing a fuel level belief to remain for the other case. It was found that the missing fuel level deletion was after the fuel level update when the agent was planning to taxi at the start of the mission. The error was fixed by adding a deed −fuel(level,F,N) to one of the plans at the point after the Executive receives a route back from the Planner:

```
+poll( self ) : {B enactRoute( taxi ,Num), B waiting(planner,enactRoute, taxi ), B fuel ( level ,F,N), B   1
    flightPhase (waitingAtRamp), B destination (D), B position (Lat,Lon,Alt ), B fuel ( level ,F,N), G
    missionComplete [achieve]} <- lock, -waiting(planner,enactRoute, taxi ), -poll( self ),
    send(planner,reqRoute,D,Lat,Lon,F), +waiting(planner,reqRoute), -fuel(level,F,N), +polled(self), unlock;
```

In order to formally verify that the error was indeed fixed, the agent was model checked successfully for the same property:

$$\Box(\mathsf{B}(\text{exec}, \text{fuel}(\text{level}, \text{L}, \text{N}) \Longrightarrow \Diamond \neg \mathsf{B}(\text{exec}, \text{fuel}(\text{level}, \text{L}, \text{N})))$$

```
Report for examples.gwendolen.uav.inputsensors16.UavAgent
====================================================== results
no errors detected
====================================================== statistics
elapsed time:       1:58:46
states:             new=365, visited=261, backtracked=625, end=0
search:             maxDepth=37, constraints=0
choice generators:  thread=2, data=364
heap:               gc=267002, new=135911979, free=135829243
instructions:       1694244565
max memory:         70MB
loaded code:        classes=281, methods=3800
=============================================================
search finished: 9/13/12 3:22 PM
```

Importantly, this error was not found during simulated flight trials using VESL, because the virtual prototype was always equipped with enough fuel for each scenario tested. The bug may have been found during simulation if, for example, VEC123 was at Sumburgh Airport (minutes from the end of its mission) without sufficient fuel to approach the airport and land. In this case, the Executive, erroneously having two different fuel level beliefs, may have used the wrong fuel level and believed that it had sufficient fuel to complete its mission.

VI. Summary & Conclusions

In this paper an approach to generating evidence towards certification of autonomous unmanned aircraft has been described based on formal model checking and simulation of agent-based autonomous systems. A rational agent-based autonomous system called the Executive was described. The Executive is designed to guide the flight of an autonomous unmanned aircraft through simulated civil airspace. This flight can be simulated within the Virtual Engineering Simulation Laboratory. The Executive has a number of subsystems which communicate with the Executive agent and provide an interface with its environment, enabling the

Executive agent to be updated to reflect the ever-changing state of the environment as well as providing a way for the Executive agent to navigate an unmanned aircraft through that environment.

In Section II the concepts of rational agents based on beliefs, desires and intentions as well as formal agent model checking were introduced. It was shown how the VEC123 autonomous unmanned aircraft virtual prototype was constructed, and how the rational agent-based autonomous system (known as the Executive agent) was built.

In Section III it was shown that requirements for an autonomous unmanned aircraft based on the Rules of the Air and Airmanship can be derived and encoded in a linear temporal logic (LTL) extended with operators to describe the beliefs, goals (i.e., desires), intentions and actions of rational agents. These formal requirements, known as properties, were then used in the formal verification of the agent-based autonomous systems for the unmanned aircraft virtual prototype VEC123. Twenty-three different properties were given and used to formally verify that the Executive satisfies the requirements concerning Rules of the Air and Airmanship on which the properties were based. These rules expand the "small subset" of Rules of the Air formally verified in earlier work by Webster et al. [8] to cover a more representative and significant subset of the Rules of the Air, as well as the somewhat nebulous concept of Airmanship, which is not defined in official documents such as the Rules of the Air [9] and must be derived from other sources, e.g., in the case of this paper, interviews with subject matter experts.

In Section IV it was shown that certain requirements based on physical quantities were difficult to model check as these quantities were not present in the "implicit" low-fidelity model of the environment in which sensor inputs are modelled only. In order to enable the model checking of these quantitative properties the low-fidelity environment model of the Executive was replaced with a higher fidelity environment model in which the physical quantities were modelled more explicitly. In particular, the positions of the unmanned aircraft and an intruder aircraft were modelled in order to model check that the "500 feet" minimum separation rule was satisfied for a detect-and-avoid manoeuvre. This higher fidelity environment model was based on an analysis of a high fidelity real-time flight simulation of a detect-and-avoid manoeuvre implemented using models of autopilot-like low-level electronic control systems. It was then shown that the higher fidelity environment model could be used in the formal verification of the quantitative requirement across a range of different sensor ranges. In order to validate the higher fidelity environment model the minimum separation distances were compared with those from the real-time flight simulation and a discrepancy was found. It was postulated that this may be due to an inaccuracy in the way the turns of the vehicle in the detect-and-avoid manoeuvre were modelled in the higher fidelity environment model. Specifically, the turns were instantaneous and not gradual, meaning that the unmanned aircraft would effectively complete its turn more quickly than the unmanned aircraft in the real-time flight simulation. The higher fidelity environment model was corrected to more accurately reflect the flight simulation and the discrepancy between the two was reduced, providing an increased level of trust in the higher fidelity environment model.

In Section V it was shown how our approach to formal verification of autonomous unmanned aircraft can be used to detect, identify and correct errors within the agent's program. This process enables virtual engineering of the autonomous unmanned aircraft through the identification and correction of errors. By executing the agent in a simulated environment, the dependence on real-world tests can be reduced. This in turn reduces the financial, social and economic costs of developing autonomous systems for practical use in aerospace applications.

Together, the different verification methods described in this paper provide a higher level of confidence in our approach to providing a means to generating evidence for certification of autonomous unmanned aircraft using formal model checking and simulation. Furthermore, this work demonstrates how formal model checking and flight simulation can be used together to provide higher levels of assurance of the safety of autonomous unmanned aircraft than they would alone. Key evidence would be given by: (i) the outputs of the model checker, (ii) higher fidelity environment model checking, and (iii) model comparison and refinement based on flight simulation. This evidence could be presented to a regulatory authority such as the CAA in the UK or the FAA in the USA for the certification of a rational agent-based autonomous control system for an unmanned aircraft. Furthermore, the approach described here is generic and could be applied to other manoeuvres and scenarios for unmanned aircraft beyond the simple detect-and-avoid scenario that we have used.

This paper is part of a larger project to investigate different means of gathering evidence (using formal methods and simulation) to present to national regulatory authorities (such as the CAA) to support certification of autonomous unmanned aircraft. Unmanned aircraft are likely to require the use of autonomous systems. Even where unmanned aircraft are remotely piloted, autonomous systems will be required in the

case where communications with a ground-based human operator have failed. The authors anticipate that over time the advantages of autonomous systems for unmanned aircraft — such as verifiability, reliability and cost — will catalyse their adoption in civil unmanned aviation and beyond.

A. Comparisons with Related Work

The work in this paper follows on from previous work by Webster et al. [8] on the use of agent model checking to formally verify selected Rules of the Air for autonomous control systems for unmanned aircraft, and extends it to cover a larger number of formally-defined requirements (a total of twenty-three) including requirements based on physical quantities. Furthermore, it is shown how the flight simulation facilities developed by Cameron et al. [10] can be used to refine the higher fidelity environment models used. An earlier version of the work in this paper can be found elsewhere [11]. A detailed comparison between this work and the earlier work can be found in Section I.

In related work, a significant subset of sixty Rules of the Air was formalised using linear temporal logic by Liu. [35]. Liu's emphasis was on the formalisation of the Rules, rather than using them for formal verification of autonomous unmanned aircraft. However the ability to formalise an even larger subset of Rules of the Air than that examined in this paper further indicates the suitability of LTL-based model checking for the formal verification of autonomous systems for use in aircraft.

A similar approach to ours is given by Bass et al. [36]., who describe formal verification of collaboration between humans and agents in aerospace applications based on model checking, simulation and abstract modelling of hybrid systems. Their work is part of the *NextGen Authority and Autonomy* project on the next generation of air traffic control procedures involving the automated exchange of information between pilots, ground controllers and automated systems in aircraft and on the ground. The broad aim is to ensure that there is no potential for unexpected behaviour in the NextGen system (known as "Automation Surprise"). Bass et al. develop models of humans and automated systems using the agent paradigm, which are model checked using an SMT (Satisfiability Modulo Theories) solver. It is demonstrated how unexpected behaviour can be discovered using the model checker for a scenario involving the Airbus A320 automated speed protection functionality. The key similarity between the work of Bass et al. and the approach described in this paper is that simulation and model checking are used together to develop a level of assurance that aircraft computer systems will behave as expected. However, the level of agent modelling differs in the two approaches; Bass et al. use an abstract model of human and machine agency, whereas in this paper the agents verified are more detailed and concrete as they are embodied in executable agent programs.

Bakera et al. [37] describe an approach to game-based model checking of autonomous systems for use in space applications. The authors apply their technique to the behaviour of the European Space Agency's planned ExoMars rover in the collection of Martian soil samples, and demonstrate model checking of the scenario. A key difference is that in our approach we examine specifically *rational agent-based* autonomous systems whereas Bakera et al. examine an autonomous system which may or may not involve an agent. Furthermore, the Agent JPF model checker used in our approach examines Java programs at the bytecode level, whereas the approach of Bakera et al. examines the behaviour of a system at a much higher level of abstraction.

Clarke et al. [38] present an approach to model checking programs with large numbers of states based on an abstraction of the program. In one case the authors are able to model check a program with over 10^{1300} states. Our approach is similar, except that we do not model check an abstraction of a program but rather the program itself. However, we do form an abstraction of the environment, as real world environments are likely to have an extremely large (if not infinite) number of states.

Platzer & Clarke [39] describe how formal verification can be applied to collision avoidance manoeuvres, in particular the "fully flyable tangential roundabout manoeuvre" (FTRM) in which two aircraft in danger of collision are able to safely manoeuvre away through the joint application of the manoeuvre. The authors use a proof assistant for non-linear hybrid systems to prove that in all cases the FTRM is collision-free. While we have used a similar example of a collision avoidance manoeuvre in this work, our work is intended to be indicative of an approach towards gathering evidence for certification of autonomous unmanned aircraft, and could be applied to other kinds of manoeuvre or scenario.

There have been many more uses of formal methods in the engineering of unmanned aircraft and spacecraft. For example: Sward used SPARK Ada to prove correctness of UAV cooperative software [40]; Barringer et al. [41] use runtime verification to formally analyse log files containing spacecraft telemetry data; Chaudemar et al. use the Event-B formalism to describe safety architectures for autonomous UAVs [42];

Jeyaraman et al. use Kripke models to model multi-UAV teams and use SPIN to verify safety and reachability properties amongst others [43]; Sirigineedi et al. use Kripke models to model UAV cooperative search missions, and use the SMV model checker to show that the UAVs do not violate key safety properties [44]. Formal methods have also been applied to autonomous systems in the aerospace domain: Pike et al. describe an approach to V&V of UAVs using lightweight domain-specific languages; Brat et al. use the PolySpace C++ Verifier and the assume–guarantee framework to verify autonomous systems for space applications [45]; while Bordini et al. proposed the use of model checkers to verify human–robot teamwork in space [46]. Importantly, none of these approaches use formal verification to establish that an autonomous systems is "equivalent" (even to a limited extent) to a human pilot, as we do here.

B. Future Work

In this paper, model checking and simulation were used to generate evidence to support certification of a rational agent-based autonomous system for an unmanned aircraft. As described in Section I, the involvement of a pilot was not made explicit, although the autonomous systems described in this paper could be adapted to work alongside a pilot. Indeed, recent guidance from the International Civil Aviation Organization (ICAO) has indicated that all unmanned aircraft should be remotely-piloted [6]. However it seems likely that some level of autonomy will always be essential in order to maintain safe operations; a remotely-piloted aircraft without any on-board autonomous systems is potentially dangerous should the communication with the pilot be lost. In the case of communications failure, a remotely-piloted aircraft will necessarily become fully autonomous, if only for a short time. Once any kind of autonomous system is used on-board an unmanned aircraft it will necessarily come under a similar level of scrutiny to other on-board avionics systems, and therefore will require certification. Therefore, in this paper we focus only on a fully autonomous virtual prototype unmanned aircraft, with the expectation that the tools and techniques used to provide evidence of the safety of fully autonomous aircraft can also be applied to the certification of partially-autonomous remotely-piloted aircraft (e.g., [47]). An essential direction for future work is to examine how the introduction of a pilot might affect the autonomous systems and verification methods described in this paper. Recent work in verification of human–robot teams [48] may provide some inspiration in this regard.

Whilst twenty-three different properties concerning Rules of the Air and Airmanship are presented in this paper, it is possible that there are still requirements which may be difficult to model accurately using linear temporal logic. For example, there are Rules of the Air that may require more expressive logics, e.g.: "... a flying machine shall move clear of the landing area *as soon as it is possible* to do so after landing." [9] The text in emphasis is a temporal statement that doesn't involve an obvious application of the eventually ($\Diamond$), always ($\Box$) or next-state ($\bigcirc$) operators. This kind of statement could be encoded in a set of rules that, for example, state that the agent in control of the unmanned aircraft will not delay in directing the aircraft to move from the landing area, for example, using the next-state operator. Another possibility might be to use a real-time model checker which permits hard restrictions on the amount of time for an operation, e.g., $\Diamond^{5s}x$ meaning x will become true within 5 seconds. Examples of model checkers with real-time capabilities include Uppaal [49] and PRISM 4.0 [50].

The work described in Section IV is a proof-of-concept of the use of higher fidelity environment models to model check quantitative requirements for autonomous systems. However, the approach can be developed and expanded in a number of ways. Firstly there are many more Rules of the Air which are based on physical quantities (a small sample is given in Section IV). One obvious extension of this work would be to develop higher fidelity environment models to enable these Rules of the Air to be model checked as well. Note that this approach is not confined to the Rules of the Air; these are used only as clear examples of quantitative requirements, and it is likely that there are many more quantitative requirements beyond the Rules of the Air.

In this paper it is assumed that (i) the requirements of the autonomous system for the unmanned aircraft are known, and (ii) that they have been accurately translated into a formal specification language such as linear temporal logic (LTL). Of course, (i) and (ii) cannot be taken for granted and would necessitate requirements engineering approaches for the reliable elicitation, specification, derivation and management of requirements. One obvious candidate would be the goal-oriented requirements engineering (GORE) methodologies, of which Knowledge Acquisition in Automated Specification (KAOS) would be particularly relevant [51].

It is worth noting that the approach described in this paper is not limited to unmanned aircraft. In fact, any autonomous system using the abstract architecture given in Fig. 2 that is embodied in a physical environment could be analysed using this approach, including unmanned ground vehicles, robots for manufacture or

assisted living, autonomous space vehicles, etc. [23]

This paper contains an example of how model checking and simulation might be used to gather evidence towards certification of an autonomous unmanned aircraft. However, the software tools used in this paper (particularly the Agent JPF model checker) would need to undergo a rigorous *tool qualification* [52] analysis before evidence generated by the tools could be accepted by a regulatory authorities like the FAA or the CAA. Model checking and other formal methods are increasingly seen as viable tools for generating evidence for certification, as can be seen in the increasing attention given to them in standards like DO-178C [53, 54], compared to its predecessor, DO-178B [52]. However this trend must be complemented by an open-minded approach by regulators and manufacturers in order to maximise the benefits of formal model checking and simulation in the certification of aircraft and other safety-critical systems.

Acknowledgments

This work is supported through the Virtual Engineering Centre (VEC), which is a University of Liverpool initiative in partnership with the Northwest Aerospace Alliance, the Science and Technology Facilities Council (Daresbury Laboratory), BAE Systems, Morson Projects and Airbus (UK). The VEC is funded by the Northwest Regional Development Agency (NWDA) and European Regional Development Fund (ERDF) to provide a focal point for Virtual Engineering research, education and skills development, best practice demonstration, and knowledge transfer to the aerospace sector. For more information see [24].

The authors would like to thank Dr. Charles Patchett of the Virtual Engineering Centre for his insightful comments on this paper.

References

[1] Weibel, R. E. and Hansman, R. J., "Safety Considerations for Operation of Unmanned Aerial Vehicles in the National Airspace System," Tech. Rep. ICAT-2005-1, MIT International Center for Air Transportation, 2005.

[2] Zaloga, S. J., Rockwell, D., and Finnegan, P., "World Unmanned Aerial Vehicle Systems: 2011 Market Profile and Forecast," Teal Group Corporation, 2011, `http://tealgroup.com/`. Accessed 2012-05-28.

[3] Civil Aviation Authority, "CAP 553 BCAR Section A: Airworthiness Procedures where the CAA has Primary Responsibility for Type Approval of the Product," `http://www.caa.co.uk/docs/33/CAP553.PDF`, October 2011, Accessed 2012-10-31.

[4] Civil Aviation Authority, "CAP 722 Unmanned Aircraft System Operations in UK Airspace — Guidance," `http://www.caa.co.uk/docs/33/CAP722.pdf`, April 2010, Accessed 2012-05-30.

[5] Wooldridge, M., *An Introduction to Multiagent Systems*, John Wiley & Sons, 2002.

[6] "Unmanned Aircraft Systems (UAS)," 2011, ISBN 978-92-9231-751-5.

[7] Jones, A., "UAS Virtual Certification," Royal Aeronautical Society Unmanned Aircraft Systems Annual Two Day Conference, October 2011, `http://www.astraea.aero/downloads/RAeS%20Conference%202011/ASTRAEA_VC_overview_RAES_2011_V1.ppt.pdf` Accessed 2012-10-10.

[8] Webster, M., Fisher, M., Cameron, N., and Jump, M., "Formal Methods for the Certification of Autonomous Unmanned Aircraft Systems," *The 30th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2011)*, edited by F. Flammini, S. Bologna, and V. Vittorini, Vol. 6894 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 228–242.

[9] Civil Aviation Authority, "CAP 393 Air Navigation: The Order and the Regulations," `http://www.caa.co.uk/docs/33/CAP393.pdf`, April 2010, Accessed 2012-06-01.

[10] Cameron, N., Webster, M., Jump, M., and Fisher, M., "Certification of a Civil UAS: A Virtual Engineering Approach," *AIAA Modeling and Simulation Technologies Conference, Portland, Oregon, Aug. 8-11, 2011*, 2011, AIAA-2011-6664.

[11] Webster, M., Cameron, N., Jump, M., and Fisher, M., "Towards Certification of Autonomous Unmanned Aircraft Using Formal Model Checking and Simulation," *Infotech@Aerospace 2012, 19–21 June 2012, Garden Grove, California*, 2012, AIAA-2011-6664.

[12] Holzmann, G., *The Spin Model Checker: Primer and Reference Manual*, AW, 2004.

[13] "Java," `http://www.java.com/` Accessed 2012-11-16.

[14] Dennis, L. A., Fisher, M., Webster, M. P., and Bordini, R. H., "Model Checking Agent Programming Languages," *Automated Software Engineering*, Vol. 19, No. 1, March 2012, pp. 5–63.

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

[16] McGrath, S., Chacón, D., and Whitebread, K., "Intelligent Mobile Agents in Military Command and Control," *Autonomous Agents 2000 Workshop/Agents in Industry, Barcelona, Spain*, 2000.

[17] Karim, S. and Heinze, C., "Experiences with the design and implementation of an agent-based autonomous UAV controller," *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, ACM, New York, NY, USA, 2005, pp. 19–26.

[18] Semmel, G. S., Davis, S. R., Leucht, K. W., Rowe, D. A., Smith, K. E., and Bölöni, L., "Space Shuttle Ground Processing with Monitoring Agents," *IEEE Intelligent Systems*, January–February 2006, pp. 68–73.
[19] Rao, A. and Georgeff, M., "BDI Agents: from Theory to Practice," *Proc. 1st International Conference on Multi-Agent Systems (ICMAS)*, San Francisco, USA, 1995, pp. 312–319.
[20] Dennis, L. A. and Farwer, B., "Gwendolen: A BDI Language for Verifiable Agents," *Logic and the Simulation of Interaction and Reasoning*, AISB'08 Workshop, 2008.
[21] Crossley, J. N., Ash, C. J., Brickhill, C. J., Stillwell, J. C., and Williams, N. H., *What is Mathematical Logic*, Oxford University Press, 1972.
[22] Dennis, L. A., Fisher, M., Lisitsa, A., Lincoln, N., and Veres, S. M., "Satellite Control Using Rational Agent Programming," *IEEE Intelligent Systems*, Vol. 25, No. 3, May/June 2010, pp. 92–97.
[23] Fisher, M., Dennis, L., and Webster, M., "Verifying Autonomous Systems," *Communications of the ACM*, 2013, in press.
[24] "Virtual Engineering Centre," `http://www.virtualengineeringcentre.com/` Accessed 2012-11-15.
[25] Webster, M., Fisher, M., Jump, M., and Cameron, N., "Model Checking and the Certification of Autonomous Unmanned Aircraft Systems," Tech. Rep. ULCS-11-001, University of Liverpool Department of Computer Science, 2011.
[26] Baier, C. and Katoen, J.-P., *Principles of Model Checking*, MIT Press, Cambridge, MA, USA, 2008, ISBN 978-0-262-02649-9.
[27] Clarke, E., Grumberg, O., and Peled, D., *Model Checking*, MIT Press, 1999.
[28] Visser, W., Havelund, K., Brat, G. P., Park, S., and Lerda, F., "Model Checking Programs," *Automated Software Engineering*, Vol. 10, No. 2, 2003, pp. 203–232.
[29] Fisher, M., *An Introduction to Practical Formal Methods Using Temporal Logic*, Wiley, 2011.
[30] "Model-Checking Agent Programming Languages," `http://mcapl.sourceforge.net` Accessed 2012-09-11.
[31] Wooldridge, M., *Reasoning about Rational Agents*, The MIT Press, 2000, ISBN 978-0262232135.
[32] Civil Aviation Authority, "CAP 493: Manual of Air Traffic Services — Part 1," `http://www.caa.co.uk/docs/33/CAP493Part1.pdf`, July 2012, Accessed 2012-10-31.
[33] Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W., and White, J. L., "Formal Analysis of the Remote Agent Before and After Flight," *The Fifth NASA Langley Formal Methods Workshop, Virginia, USA*, 2000.
[34] Advanced Rotorcraft Technology, Inc., "FLIGHTLAB," `http://www.flightlab.com/flightlab.html`. Accessed 2012-05-28.
[35] Liu, D., *Formalizing Rules of the Air for Formal Verification of Autonomous UAS*, Master's thesis, University of Liverpool, Department of Computer Science, 2012.
[36] Bass, E. J., Feigh, K. M., Gunter, E., and Rushby, J., "Formal Modeling and Analysis for Interactive Hybrid Systems," *Proceedings of the Fourth International Workshop on Formal Methods for Interactive Systems (FMIS 2011)*, edited by J. Bowen and S. Reeves, Electronic Communications of the EASST, 2011, ISSN 1863-2122.
[37] Bakera, M., Margaria, T., Renner, C. D., and Steffen, B., "Game-Based Model Checking for Reliable Autonomy in Space," *Journal of Aerospace Computing, Information, and Communication*, Vol. 8, No. 4, 2011, pp. 100–114.
[38] Clarke, E. M., Grumberg, O., and Long, D. E., "Model Checking and Abstraction," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, 1994, pp. 1512–1542.
[39] Platzer, A. and Clarke, E. M., "Formal Verification of Curved Flight Collision Avoidance Maneuvers: A Case Study," *FM 2009: Formal Methods Second World Congress*, edited by A. Cavalcanti and D. R. Dams, Vol. 5850 of *Lecture Notes in Computer Science*, Springer, 2009.
[40] Sward, R. E., "Proving Correctness of Unmanned Aerial Vehicle Cooperative Software," *Proc. IEEE International Conference on Networking, Sensing and Control*, 2005.
[41] Barringer, H., Groce, A., Havelund, K., and Smith, M., "Formal Analysis of Log Files," *Journal of Aerospace Computing, Information, and Communication*, Vol. 7, 2010, pp. 365–390.
[42] Chaudemar, J.-C., Bensana, E., and Seguin, C., "Model Based Safety Analysis for an Unmanned Aerial System," *Proc. Dependable Robots in Human Environments (DRHE)*, 2010.
[43] Jeyaraman, S., Tsourdos, A., Zbikowski, R., and White, B., "Formal Techniques for the Modelling and Validation of a Co-operating UAV Team that uses Dubins Set for Path Planning," *Proc. American Control Conference*, 2005.
[44] Sirigineedi, G., Tsourdos, A., Zbikowski, R., and White, B. A., "Modelling and Verification of Multiple UAV Mission Using SMV," *Proc. FMA-09*, Vol. 20 of *EPTCS*, 2009.
[45] Brat, G., Denney, E., Giannakopoulou, D., Frank, J., and Jonsson, A., "Verification of Autonomous Systems for Space Applications," *Proc. IEEE Aerospace Conference*, 2006.
[46] Bordini, R. H., Fisher, M., and Sierhuis, M., "Formal Verification of Human-Robot Teamwork," *Proc. 4th Int. Conf. Human-Robot Interaction (HRI)*, ACM, 2009, pp. 267–268.
[47] Taylor, R. M., "Capability, Cognition and Autonomy," *RTO HFM Symposium on The Role of Humans in Intelligent and Automated Systems (RTO-MP-088)*, NATO, 2002, Warsaw, Poland, 7-9 October 2002.
[48] Stocker, R., Dennis, L. A., Dixon, C., and Fisher, M., "Verification of Brahms Human–Robot Teamwork Models," *Proc. 13th European Conference on Logics in Artificial Intelligence (JELIA-2012)*, Vol. 7519 of *Springer Lecture Notes in Computer Science*, 2012, pp. 385–397.
[49] Larsen, K. G., Pettersson, P., and Yi, W., "UPPAAL in a Nutshell," *Int. Journal on Software Tools for Technology Transfer*, Vol. 1, No. 1–2, Oct. 1997, pp. 134–152.

[50] Kwiatkowska, M., Norman, G., and Parker, D., "PRISM 4.0: Verification of Probabilistic Real-time Systems," *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, edited by G. Gopalakrishnan and S. Qadeer, Vol. 6806 of *LNCS*, Springer, 2011, pp. 585–591.

[51] Lapouchnian, A., "Goal-Oriented Requirements Engineering: An Overview of the Current Research," Tech. rep., University of Toronto, 2005, `http://www.cs.toronto.edu/~alexei/pub/Lapouchnian-Depth.pdf`. Accessed 2013-05-25.

[52] RTCA Inc., "DO-178B: Software Considerations in Airborne Systems and Equipment Certification," December 1992.

[53] RTCA Inc., "DO-178C: Software Considerations in Airborne Systems and Equipment Certification," December 2011.

[54] RTCA Inc., "DO-333: Formal Methods Supplement to DO-178C and DO-278A," December 2011.