PRELIMINARY APPLICATION OF FORMAL VERIFICATION TO AN AUTONOMY ARCHITECTURE FOR UNMANNED AIRCRAFT

Laura R. Humphrey[1] & César A. Muñoz[1]

[1]NASA Langley Research Center, Hampton, VA, USA

Abstract

There is a desire to design autonomous systems in such a way that capabilities can be easily added or re-combined to produce new behaviors while preserving their safety properties. ICAROUS, a prototype software architecture for building safety-centric autonomous unmanned aircraft applications, is designed to support this type of extensibility and re-configurability. In ICAROUS, core capabilities are implemented as individual software services or modules, so that enabling access to new capabilities simply requires adding new modules. To make use of these capabilities, ICAROUS includes a specialized module that provides a general framework for configuring the relative priorities, conditions, and rules that govern how different modules should be engaged and disengaged during flight. The inherent complexity of coordinating multiple modules under changing conditions makes it difficult to determine whether a particular configuration could have erroneous behaviors in certain circumstances. A robust set of integration tests can help discover errors, but testing can only realistically cover a relatively small proportion of total system behaviors. Developing good tests and interpreting the results to pinpoint the cause of errors when they arise can also be very time-consuming. To supplement testing, formal methods can be used to model and analyze systems, achieving better coverage and simplifying the process of finding, understanding, and fixing errors. To demonstrate these benefits, this paper explores the application of formal methods to ICAROUS. In particular, the Spin model checker is used to specify requirements for and model portions of the system, then verify whether the model satisfies the requirements and find and fix errors when it does not.

Keywords: Formal Methods, Formal Verification, Model Checking, Unmanned Aircraft Systems, Autonomy Stack

1. Introduction

The NASA Air Mobility Pathfinders (AMP) project aims at the development of a reference architecture for safe, secure, and scalable Urban Air Mobility (UAM) operations. Toward this end, the project is building software prototypes and simulation infrastructure to investigate key airspace integration issues of midterm UAM operations. The Independent Configurable Architecture for Reliable Operations of Unmanned Systems (ICAROUS) is one such software prototype. ICAROUS provides a flexible framework for integrating new capabilities that also includes a set of baseline capabilities to monitor and enforce safety constraints such as detect and avoid (DAA) and geofencing. ICAROUS is implemented as service-oriented architecture consisting of multiple interacting software services or modules. This type of architecture makes it easy to add new capabilities, but it becomes more challenging to validate and verify its behavior. Since ICAROUS simulations will inform the development of AMP's architecture, validating that it behaves as expected is critical to producing relevant data.

Integration testing is commonly used to check for unsafe or erroneous interactions between modules, but testing can only realistically cover a relatively small proportion of total system behaviors. Formal methods, i.e., mathematically-based tools and approaches for software and hardware verification, can be used to fully analyze or prove properties of system models. Formal methods can often find

errors that are not found through testing and have been successfully applied to several aerospace and autonomous systems [1].

This paper explores the application of formal methods to ICAROUS by modeling some of its key modules in the Spin model checker. Spin has been used in applications such as the Mars Science Laboratory mission [2] and Deep Space 1 mission [3] to identify and correct errors in the design of several critical software components. It can be used both to simulate models, i.e., provide random execution traces, and to verify whether any behaviors of a model can result in deadlock or non-progress cycles or can violate system-level specifications written in temporal logic. If verification finds a violation, Spin provides a counterexample that demonstrates how the violation can occur, which is useful for identifying and fixing the underlying cause.

The rest of this paper is organized as follows. Background on model checking and Spin is provided in Section 2. A brief description of ICAROUS is given in Section 3. Section 4 walks through the iterative process of modeling and verifying a minimal configuration of ICAROUS to identify and propose a fix to a known issue. Section 5 concludes with a summary and discussion of future work.

2. Background on the Spin Model Checker

This section starts by providing background on model checking. Then it provides background on the Spin model checker specifically.

2.1 Model Checking

Model checking is a formal technique that explores all possible behaviors of a model whose state changes over time to determine whether the model is guaranteed to satisfy a given specification. The types of models used in model checking are generally in terms of states, what logical propositions hold in each state, what operations are allowed in each state, and what effects those operations have in terms of transitions to new states and changes to which propositions hold.

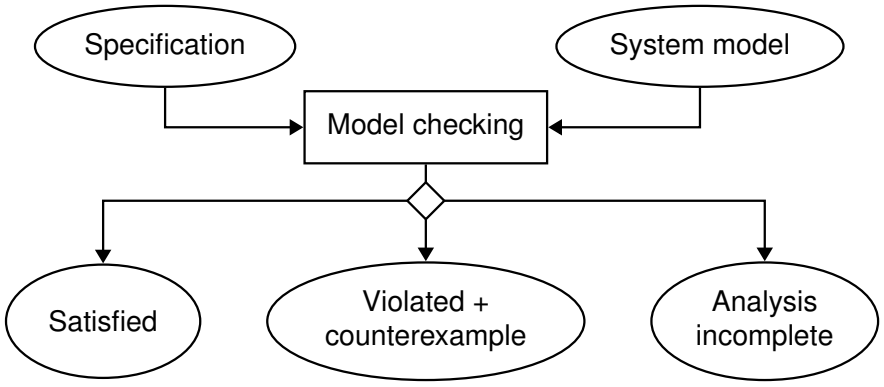

Figure 1 – The model checking process.

Figure 1 illustrates the model checking process. A model checker takes as input a specification and a system model and attempts to verify whether all behaviors of the model satisfy the specification. If the model checker fully explores the model and finds no violations, then the model is guaranteed to always satisfy the specification. If the model checker finds a violation, it provides a counterexample that shows how the model can violate the specification. If the model checker is unable to fully explore the model and does not find a counterexample, then it is unknown whether the model is guaranteed to satisfy the specification. Errors resulting in counterexamples can come from several different sources. They can be specification errors, i.e., the specification does not correctly capture what the specifier intended to express and needs to be revised. They can be modeling errors, i.e., the model does not correctly capture the system designer's intent and needs to be revised. They can also be design errors, i.e., the model accurately represents the system but there is a flaw in the system design.

There are different types of model checkers that are specialized toward different types of models and specifications. For example, there are several model checkers for modeling finite-state systems and verifying whether they satisfy specifications written in temporal logic that describe the relative ordering of properties that should hold over the course of execution of the model. These include model checkers such as NuSMV [4], Spin [5], and Maude LMC [6]. Other model checkers such as Kind 2 [7] target infinite-state systems with real-valued variables. Model checkers such as PRISM [8] are suitable for modeling systems with behaviors that can be characterized by probability distributions and verifying that they satisfy temporal logic specifications with a given probability. There are also model checkers such as Uppaal [9] for modeling systems with real-time constraints.

This paper uses Spin, a finite-state model checker. Models in Spin can be represented mathematically as finite-state transition systems. A *transition system* TS is a tuple $TS = (S, S_0, \rightarrow, AP, L)$ where

- S is a set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $\rightarrow \subseteq\ S \times S$ is a state transition relation,
- AP is a set of atomic propositions,
- $L : S \rightarrow 2^{AP}$ is a labeling function that indicates which propositions hold in a state.

TS is called *finite* if S and AP are finite.

To summarize, a transition system describes the ways in which the state of a system changes over time, with atomic propositions corresponding to simple statements that are either true or false depending on the state. For example, in a transition system that models execution of a simple single-threaded computer program, each state would represent the values of all program variables along with the current value of the program counter. Examples of atomic propositions are $x > 0$ or $x < 1000$ for an integer x in the program. For a given state s, $L(s) \subseteq 2^{AP}$ (where 2^{AP} is the power set of AP) is the set of all atomic propositions that are true in that state. Let s_i indicate the state of the system at time step i for $i >= 0$. A transition system produces sequences of states called *paths* of the form $s_0 s_1 s_2 \ldots$ where $s_0 \in S_0$ and each $(s_i, s_{i+1}) \in \rightarrow$. Note that the transition relation $\rightarrow$ can contain multiple transitions out of each state, so there can be multiple valid paths starting from the same initial state s_0. A *trace* of a path is a sequence of sets of atomic propositions of the form $L(s_0)L(s_1)L(s_2)\ldots$ that hold over the path according to the labeling function L.

Specifications in Spin are written in linear temporal logic (LTL). LTL formulas are formed according to the grammar:

$$\varphi ::= \text{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2.$$

LTL formulas include the standard propositional logic operators: $\wedge$ "and," $\neg$ "not," and derived operators such as $\vee$ "or," $\rightarrow$ "implies," and $\leftrightarrow$ "is equivalent to." LTL formulas also include the temporal operators $\bigcirc$ "next" and U "until." In the current state, formula $\bigcirc\varphi$ holds if φ holds in the next state in the trace, and formula $\varphi_1 \,\mathsf{U}\, \varphi_2$ holds if there is some future state in the trace for which φ_2 holds and φ_1 holds in all states until then. From these, we can also derive the $\Box$ "always" and $\Diamond$ "eventually" temporal operators. In the current state, formula $\Box\varphi$ holds if φ holds in the current and all future states, and formula $\Diamond\varphi$ holds if φ holds in some future state. Specifically, these operators are defined as $\Diamond\varphi \stackrel{\text{def}}{=} \text{true}\,\mathsf{U}\,\varphi$ and $\Box\varphi \stackrel{\text{def}}{=} \neg(\Diamond\neg\varphi)$. Spin can verify whether all traces produced by a transition system satisfy a set of specifications written in LTL and if not, return a corresponding violating path as a counterexample.

2.2 The Spin Model Checker

Spin is a model checker geared toward the design and verification of systems consisting of multiple asynchronous interacting processes. Spin models are written in Promela, a relatively small modeling

language with an intuitive, program-like notation. Promela includes constructs for modeling variables, processes, and channels used to communicate data between processes.

Basic variable types include *bit*, *bool*, *byte*, *short*, *int*, and arrays of other variable types. Message types can be defined with keyword *mtype:[name]*, which can also be used to define enumerations, since an *mtype* definition simply associates *n* given symbolic names to sequences of byte values from 1 to *n*. Custom struct-like data types can be defined with keyword *typedef*. Variables can be local to a specific process or globally accessible to all processes, the latter enabling modeling of concurrent threads of execution that share memory.

Processes consist of sequences of discrete, executable statements that determine the behavior of the model. Statements include expressions, variable declarations, and assignments, *goto* jumps, *if* selections, *do* loops, *skip* no-ops, and *unless* exception handling routines. Expressions include the standard arithmetic, comparison, increment and decrement, bit shift, and bitwise operators for integer types and the standard Boolean operators for Boolean types. The notation for these operators is the same as in the C programming language, e.g., $<$ for "less than," $++$ for "increment," $|$ for "bitwise or," and $||$ for "logical or." Statements can either be enabled or blocked, i.e., able to be executed or not. Some statements are always enabled, for example assignments to variables. Other statements can be blocked depending on the values of their variables, for example a Boolean expression that evaluates to false or an attempt to read a channel that is empty. Note that some statements, e.g., *if* selections and *do* loops, are composed of multiple constituent statements. Such statements are enabled if at least one of their constituent statements is enabled. If more than one constituent statement is enabled, then the choice of which one to execute is made non-deterministically. When a statement is blocked, the corresponding process is blocked until some other process makes a change that enables it. Each process has a process counter that tracks the next statement to be executed. When multiple active processes have enabled statements, the choice of which process executes its next statement is made non-deterministically. In other words, processes execute concurrently and asynchronously.

Channels can be synchronous or asynchronous. Synchronous channels model rendezvous ports that do not have a buffer to store messages, i.e., they require both a sending and receiving process to simultaneously handshake over the channel, otherwise an attempt to read or write on the channel is blocking. Asynchronous channels model non-blocking message passing over a buffer of a specified finite size. Attempts to write to asynchronous channels are only blocking if the channel is full and the option to drop messages on full channels has not been set. For synchronous channels, the basic notation is *[channelName]![msgType], [var1, ..., varN]* to send the values of one or more variables over a channel and *[channelName]?[msgType], [var1, ..., varN]* to receive those values from the channel and store them into one or more variables. The value of *msgType* must be a positive constant that is the same for both the send and receive operation, conventionally an element of an *mtype* definition. For asynchronous channels, this constant is not required. There are also additional functions for checking how many messages are stored on an asynchronous channel and whether the channel buffer is full or empty.

3. ICAROUS

NASA's ICAROUS[1] is an on-board prototype system that integrates mission specific software modules and highly assured core software modules for building safety-centric autonomous unmanned aircraft applications [10]. Functionally, ICAROUS is an instance of a service-oriented architecture for UAS [11] that provides services to an aircraft, called the *ownship*, through a communication layer (Figure 2). These services monitor ownship and traffic states and autonomously maneuver the ownship to enforce constraints such as aircraft separation, geofencing, path conformance, and merging and spacing. The core modules provided by ICAROUS include implementations of formally verified

[1] `https://shemesh.larc.nasa.gov/fm/ICAROUS/`.

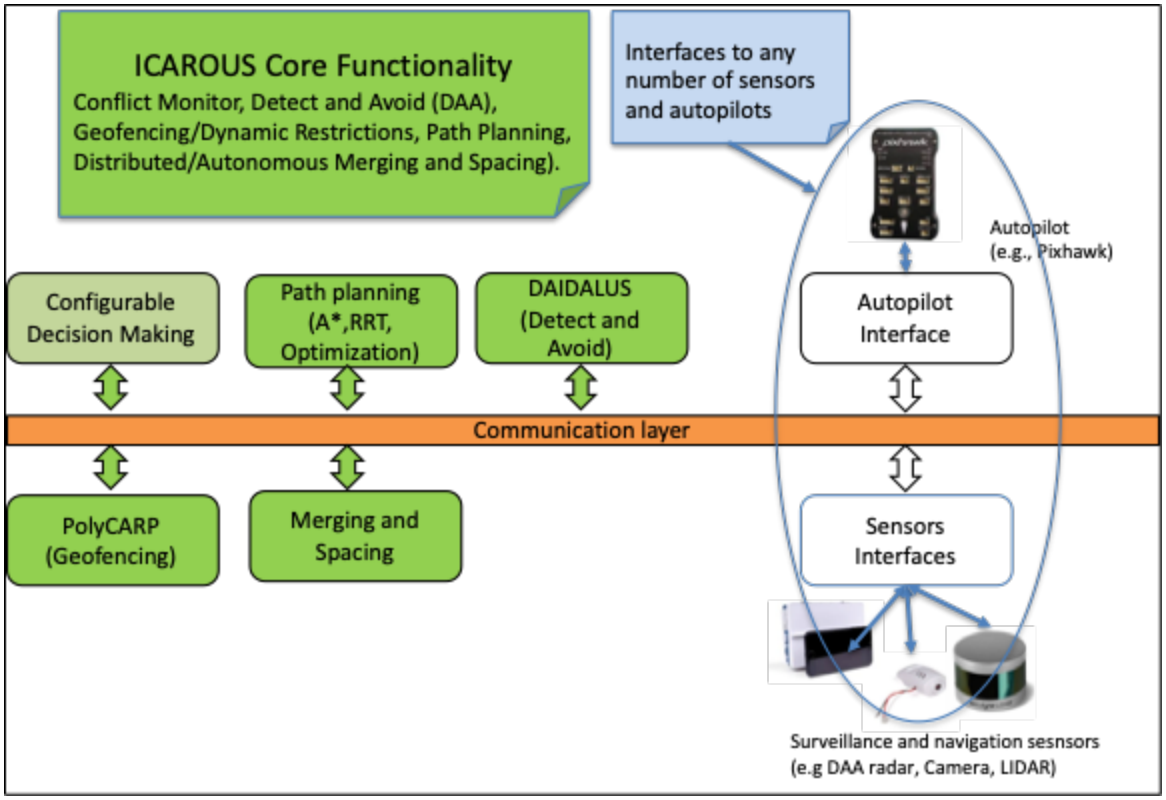

Figure 2 – ICAROUS as a service oriented architecture.

detect and avoid (DAIDALUS)[2] [12] and geofencing (PolyCARP)[3] [13] algorithms.

For communication and execution, ICAROUS relies on the core Flight System (cFS)[4], an open-source middleware developed by NASA for critical embedded software systems. Each ICAROUS module is implemented as a cFS application that communicates with other cFS applications through a software bus using a publisher-subscriber protocol. In flight, ICAROUS uses the MAVLink communication protocol[5] for sending commands to and receiving telemetry from the autopilot. Although ICAROUS was originally designed as an on-board system for small UAS, it has been integrated into a fast-time simulation environment for urban air mobility (UAM) operations [14]. Since cFS is not suitable for fast-time simulation, a Python-based simulation framework called Pycarous was developed. Pycarous mimics cFS by providing communication and synchronization capabilities, but it also generates ownship and traffic aircraft states using different flight dynamics models and simulates ground station commands. While real-time vehicle/hardware in the loop (HITL) executions are supported by cFS, fast-time simulation relies on Pycarous.

The data-flow architecture of ICAROUS is shown in Figure 3. Periodic updates of ownship and traffic states are received by the Traffic Monitor, which predicts impending loss of well clear, and the Trajectory Manager, which does path planning around geofences. These applications communicate with Cognition, a decision-making application. The decision-making logic is structured based on the flight phases of the ownship, i.e., takeoff, climb, cruise, approach, and landing. This leads to a hierarchical set of finite state machines that are driven by the outputs of the various conflict monitors. The finite state machines trigger the Guidance application, which in turn sends flight commands to the vehicle.

The state machines that govern the decision-making logic in Cognition are implemented using an

[2] https://shemesh.larc.nasa.gov/fm/DAIDALUS/.
[3] https://shemesh.larc.nasa.gov/fm/PolyCARP/.
[4] https://cfs.gsfc.nasa.gov/.
[5] https://mavlink.io/en/

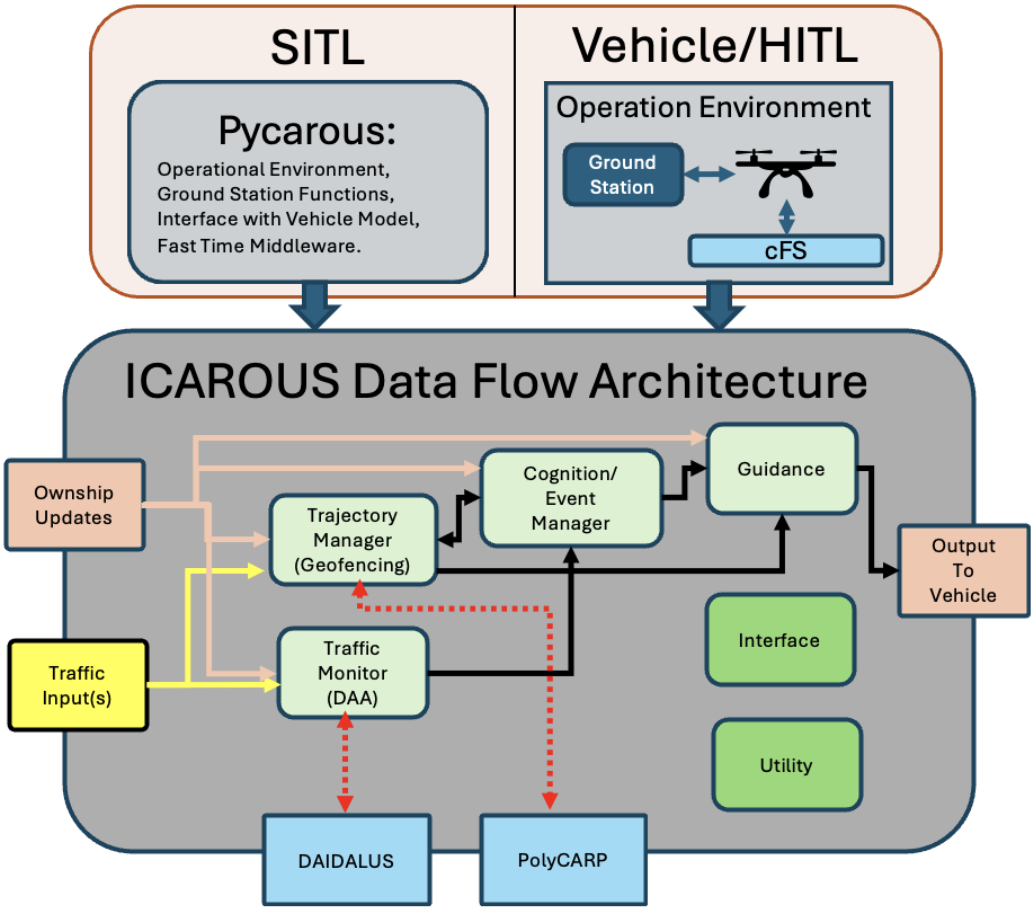

Figure 3 – ICAROUS data flow architecture.

Event Manager that allows for the definition of triggers and handlers. Each handler is associated with a Boolean trigger. The Event Manager periodically evaluates the triggers to determine which are currently true. If more than one trigger is true, it determines which corresponding handler has the highest priority. If a new handler has the highest priority, the old one is turned off and the new one is run. Therefore, at any time, only one handler is active. Running a handler requires checking the handler's current state, running the handler function that should be executed in that state, and depending on the outcome of the function, determining which state the handler should transition to next.

4. Applying Spin to ICAROUS

This section describes the modeling and verification of ICAROUS as it runs with Pycarous in Spin.

4.1 Modeling ICAROUS in Spin

ICAROUS is a complex system with many components. Modeling and verifying the entire system all at once would be a daunting task. A more reasonable approach is to model and verify the system incrementally, in this case starting with a minimal subset of its components. This allows the modeler to develop good modeling approaches tailored to the system and to build confidence in a base-level model without getting bogged down in too many details. It is also makes it easier to interpret verification results, since the model is smaller, and to focus on finding and fixing errors in targeted areas of the system, which provides a good foundation for modeling the rest of the system later.

Toward that end, the Spin model of ICAROUS presented here includes only three components: Cognition, Guidance, and Pycarous, which includes a module called the Autonomy Stack that enables interactions between ICAROUS and the rest of Pycarous. It does not include other ICAROUS modules such as the Traffic Monitor or the Trajectory Manager. It also does not focus on low-level details such as the exact computation of waypoint locations for flight plans, instead focusing on control flow and higher-level decision-making. It also instantiates certain parameters of the system with specific values, for example the length of the flight plan. While there are advanced approaches for model checking certain classes of parameterized systems [15], standard approaches require such parameters to be given fixed values. Here, the module logic does not fundamentally change based on specific values of parameters such as length of the flight plan, so a model in which they are fixed can still be used to build significant confidence in the system design.

ICAROUS is written in C++, Pycarous in Python, and the Autonomy Stack in Cython, a programming language used to interface Python code with C/C++ code. The general modeling strategy is to model function calls and methods of classes as inline procedures in Spin, unless they are very straightforward; for example, methods to set and get attributes of a class are simply modeled by modifying the attributes directly. The use of inline procedures to model functions and methods makes it easier to check conformance of the model with the code. Note that inline procedures in Spin can take arguments but cannot return values, so for functions, the last argument of an inline procedure is used to model the return value. Functions that are simple Boolean expressions are defined with macros instead. Methods of classes for which there is only one object are not modeled as taking arguments. Instead, there is a global variable of the appropriate type for each class attribute, whose name is prefixed by the class name. Classes for which there are multiple objects of that class (or its subclasses) are handled differently. Macros are used to associate names to each unique object, and these are used to look up the value of that object's attribute. This enables modeling of dynamic dispatching to process different types of handlers in Cognition's Event Manager. Enumerations in the code are modeled with *mtype* declarations and structs with *typedef* declarations. Names are kept close to those used in the code, with some exceptions for brevity or to increase readability.

Significant portions of this basic ICAROUS Spin model are shown in Listings 1 through 7. Listing 1 shows enumerations, type definitions, macros, and some of the global variables used in the model. Note that there is a single global variable *activePlanSize* for length of the flight plan. Since it is hard-coded to a set value of six and does not change, this variable is used across the modeled

modules rather than each having its own version. No other details of the flight plan are considered in this simple model. The rest of the global variables model attributes of the main Autonomy Stack, Guidance, and Cognition classes, with each set sometimes referred to here as the "state" of the corresponding module. Many are explicitly initialized, and those that are not default to 0. Note that asynchronous channel *cogState_cognitionCommands* is used for convenience to model a list.

Pycarous drives execution of the whole system, so it is modeled with a *do* loop in the *init* process shown in Listing 2. To interact with ICAROUS, Pycarous makes calls to the Autonomy Stack at every simulation time step. In the code, the first few calls update the state of the Autonomy Stack with the most recent states of all the UASs in the simulation. However, detailed UAS information is not included in this model. Here, it would only be used by Guidance to determine whether the ownship has reached a waypoint, so instead, the number of time steps needed to reach the next waypoint is modeled non-deterministically in Guidance portions of the model. The next few calls in the Pycarous code cause the Autonomy Stack to run ICAROUS by calling its own methods *Stack_RunGuidance()* and *Stack_RunCognition()*. If the simulation is just starting, Pycarous also calls an Autonomy Stack method that simply sets Cognition's *missionState* attribute to indicate that the system is in the pre-mission phase. Pycarous stops simulating once the Autonomy Stack's state indicates that the ownship has started to land.

Listing 3 shows procedures that model Autonomy Stack methods *Stack_RunGuidance()* and *Stack_RunCognition()*. Procedure *Stack_RunGuidance()* checks the Autonomy Stack's *guidanceMode* attribute. If it corresponds to no operation, then it skips to the end of the procedure. If it corresponds to takeoff, then it sets Cognition's *takeoffState* attribute to indicate that takeoff is complete and skips to the end of the procedure. Otherwise it calls *RunGuidance()*, the main method of the Guidance class, which updates Guidance attribute *nextWpId* if the most recent waypoint was reached. If this index is larger than the one stored in the Autonomy Stack, then the Autonomy Stack updates its corresponding attribute and Cognition's as well. The method *Stack_RunCognition()* calls the *RunCognition()* method of Cognition. Based on Cognition's state afterward, the Autonomy Stack checks whether the ownship has started to land, and if so, it updates its own state to record the landing. Otherwise, it loops over and processes any commands generated by Cognition. If a command is for a flight plan change, it updates its state to set the guidance mode accordingly and its next waypoint index to the one stored in the command, then it calls the *SetGuidance()* method to update the same parts of Guidance's state. If the command is for takeoff, the Autonomy Stack simply updates its own guidance mode accordingly.

Listing 4 shows portions of the model that cover Guidance. Since geofences and other traffic are not considered in the simple model developed here, its logic is relatively simple. Guidance can be in several different modes corresponding to takeoff, following a flight plan, landing, and doing nothing. Procedure *RunGuidance()* determines what Guidance does in each of these modes. Pycarous does not do any complex modeling of takeoff, so the Guidance logic for takeoff essentially does nothing except log that takeoff occurred. The Guidance logic for following a flight plan tracks the index of the next waypoint the UAS should fly to and monitors the state of the ownship to determine when it reaches it. The call to *ComputePlanGuidance()* in this mode checks whether the next waypoint index has gone beyond the end of the flight plan and if not, it calls *CheckWaypointArrival()* to check whether the next waypoint has been reached. It then checks whether the next waypoint index is the last in the flight plan and if so, it changes its mode to landing. As previously mentioned, low-level details such as the exact UAS position are not considered in this model, so this procedure non-deterministically models that it takes between one and three time steps to reach the next waypoint. Similarly, the logic for the landing mode models that it non-deterministically takes between one and three time steps to land.

Before covering portions of the model that cover Cognition, it is important to understand how handlers in the Event Manager work. In the code, each handler is a subclass of the main handler class, which has an attribute to store the handler's internal state. Possible values for each handler's state include *NOOP* (not yet operating), *INITIALIZE*, *EXECUTE*, *TERMINATE*, and *DONE*. The handler class also has a *RunEvent()* method, and each subclass provides its own implementation of three

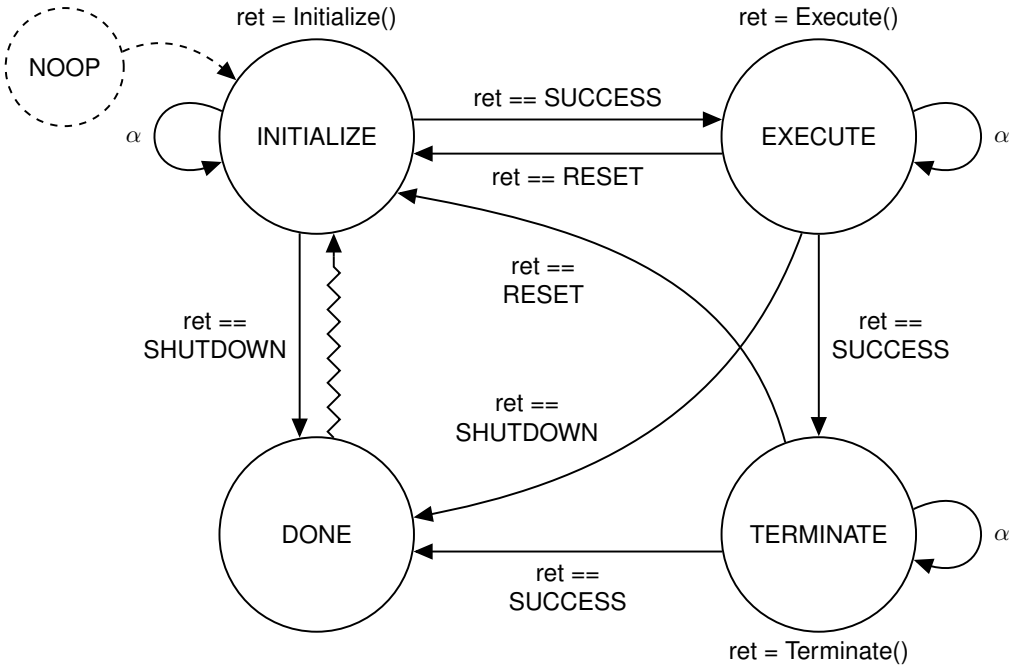

Figure 4 – The state machine logic encoded by handler method *RunEvent()*.

"event" methods: *Initialize()*, *Execute()*, and *Terminate()*. Each event method returns a value after execution: *SUCCESS*, *RESET*, *INPROGRESS*, or *SHUTDOWN*. Procedure *RunEvent()* models the state machine logic used to execute the event method that corresponds to the current state and transition to the next state based on the event method's return value. It is called by the Event Manager whenever Cognition requests that it run the appropriate event handler. For brevity, its logic is not given in any of the Listings but is shown in Figure 4. An α self-transition in a state corresponds to the default transition when no other transition out of the state is possible. Also note that the transition from *DONE* to *INITIALIZE* occurs immediately after *DONE* is reached, i.e., during the same call to *RunEvent()*. Finally, note that there is no logic in *RunEvent()* to transition from *NOOP* to *INIT*. This is done elsewhere in the Event Manager.

Listing 5 shows portions of the model that define the triggers and logic for running the top priority active handler and stopping a lower priority handler if necessary. The global array *events* models the attribute of the Event Manager class that tracks which triggers are true, indexed by the corresponding event handler's name. Procedure *RunEventMonitors()* models a loop in the code that evaluates the trigger for each handler and stores the result in *events*. After each trigger is evaluated, procedure *RunEventMonitors()* is called. If the corresponding handler was not already active, it sets the handler's state to *NOOP*. Then if there are other active handlers, it sets Cognition's state to record that the new handler is active and checks whether the new one has higher priority. If so, it sets the old one's state to *TERMINATE*, runs it, then sets its state to *DONE*. If there were no other active handlers, then it simply sets Cognition's state to record that the new handler is active. Note that the logic for *GetTopPriorityActiveHandler()* is not shown; it returns a result based on the handler for landing having the highest priority, the handler for engaging the nominal flight plan having the next highest priority, and the handler for takeoff having the lowest priority.

Listing 6 shows portions of the model that cover the event handlers. Global array *execState* models the attribute of the Event Handler class and its subclasses that holds the handler's state, indexed by handler name. Procedure *SetGuidanceMode()* models a method of Cognition used in one of the handler methods that sends a flight plan change command. The rest of the procedures model the non-trivial *Initialize()*, *Execute()*, and *Terminate()* methods of each handler. These modify Cognition's state to record events like takeoff, flight, and landing, and they construct commands that Cognition sends to other modules. They also return a result that is used by the Event Manager to determine

the next state of the handler. Handler methods not shown simply return *SUCCESS*.

Listing 1: Type definitions, macros, and global variables for the ICAROUS model.

```
mtype:guideMode_e =
 {GUIDE_NOOP, FLIGHTPLAN, TAKEOFF,
  LAND}
mtype:execState_e =
 {NOOP, INITIALIZE, EXECUTE,
  TERMINATE, DONE}
mtype:retVal_e =
 {SUCCESS, RESET, INPROGRESS, SHUTDOWN}
mtype:cmd_e =
 {TAKEOFF_COMMAND, FP_CHANGE,
  LAND_COMMAND}
mtype:takeoffState_e =
 {TAKEOFF_INACTIVE, TAKEOFF_INPROGRESS,
  TAKEOFF_COMPLETE}
mtype:missionStart_e =
 {PRE_MISSION, LAUNCH, FLIGHT, LANDING}

typedef command_t {
 mtype:cmd_e commandType;
 byte wpIndex = 0;
};

#define TAKEOFF_PHASE_HANDLER 0
#define ENGAGE_NOMINAL_PLAN 1
#define LAND_PHASE_HANDLER 2

#define IsEmpty_activeHandlers
(!(cogState_activeHandlers
    [TAKEOFF_PHASE_HANDLER] ||
    cogState_activeHandlers
    [ENGAGE_NOMINAL_PLAN] ||
    cogState_activeHandlers
    [LAND_PHASE_HANDLER]))

// Flight plan length
byte activePlanSize = 6;

// AutonomyStack class variables
bool stackState_land = false;
byte stackState_nextWP1 = 1;
mtype:guideMode_e
 stackState_guidanceMode = GUIDE_NOOP;

 // Guidance class variables
mtype:guideMode_e
 guideState_mode = GUIDE_NOOP;
byte guideState_nextWpId;
bool guideState_wpReached = false;

// Cognition class variables
mtype:missionStart_e
 cogState_missionStart = LAUNCH;
mtype:takeoffState_e
 cogState_takeoffState =
  TAKEOFF_INACTIVE;
bool cogState_nominalPlanEngaged =
 false;
byte cogState_nextWpId = 0;
bool cogState_activeHandlers[4];
chan cogState_cognitionCommands =
 [3] of { command_t };
bool cogState_icReady = false;
```

Listing 2: The *init* process, which models Pycarous.

```
init{
 bool simState_missionStarted = false;
 do
  :: !stackState_land ->
     Stack_RunGuidance();
     Stack_RunCognition();
     if
      :: !simState_missionStarted ->
         cogState_missionStart =  PRE_MISSION;
         simState_missionStarted = true;
      :: simState_missionStarted -> skip;
     fi
  :: else -> break;
 od
}
```

Listing 3: Inline procedures that model methods of the main Autonomy Stack class.

```
inline Stack_RunGuidance() {
 printf("Stack_RunGuidance()\n");
 if
  :: stackState_guidanceMode == GUIDE_NOOP ->
     goto Ret1;
  :: stackState_guidanceMode == TAKEOFF ->
     cogState_takeoffState = TAKEOFF_COMPLETE;
     goto Ret1;
  :: else -> skip;
 fi

 RunGuidance();
 byte nextWP = guideState_nextWpId;

 if
  :: stackState_nextWP1 < nextWP ->
     cogState_nextWpId = nextWP;
     stackState_nextWP1 = nextWP;
  :: else -> skip;
 fi

Ret1: skip; }

inline Stack_RunCognition() {
 printf("Stack_RunCognition()\n");
 RunCognition();

 if
  :: cogState_missionStart == LANDING ->
     stackState_land = true;
  :: else -> stackState_land = false;
 fi

 do
  :: len(cogState_cognitionCommands) > 0 ->
     command_t cmd;
     cogState_cognitionCommands?cmd;
     if
      :: cmd.commandType == FP_CHANGE ->
         stackState_guidanceMode = FLIGHTPLAN;
         stackState_nextWP1 = cmd.wpIndex;
         SetGuidanceMode(FLIGHTPLAN,
                         cmd.wpIndex);
      :: cmd.commandType == TAKEOFF_COMMAND ->
         stackState_guidanceMode = TAKEOFF;
      :: else -> skip;
     fi
  :: else -> break;
 od
}
```

Listing 4: Model of Guidance methods.

```
byte wpSteps = 0;
byte landSteps = 0;

inline SetGuidanceMode(mode, nextWP) {
 guideState_mode = mode;
 guideState_nextWpId = nextWP;
}

inline CheckWaypointArrival() {
 printf("CheckWaypointArrival()\n");
 if
  :: wpSteps < 3 ->
     wpSteps = wpSteps + 1;
     guideState_wpReached = false;
  :: wpSteps > 0 -> wpSteps = 0;
     guideState_wpReached = true;
     guideState_nextWpId =
      guideState_nextWpId + 1;
 fi;
}

inline ComputePlanGuidance() {
 printf("ComputePlanGuidance()\n");
 if
  :: guideState_nextWpId >=
      activePlanSize ->
     guideState_wpReached = true;
     goto CWA_Ret;
  :: else -> skip;
 fi

 CheckWaypointArrival();

CWA_Ret: skip;
}

inline RunGuidance () {
 printf("RunGuidance()\n");
 if
  :: guideState_mode == TAKEOFF ->
     printf("mode TAKEOFF\n");
     skip;
  :: guideState_mode == FLIGHTPLAN ->
     printf("mode FLIGHTPLAN\n");
     ComputePlanGuidance();
     if
      :: guideState_nextWpId > 1 &&
          guideState_nextWpId ==
           activePlanSize - 1 ->
         guideState_mode = LAND;
      :: else -> skip;
     fi
  :: guideState_mode == LAND ->
     printf("mode LAND\n");
     if
      :: landSteps < 3 ->
         landSteps = landSteps + 1;
      :: landSteps > 1 ->
         guideState_nextWpId =
          activePlanSize;
     fi
  :: guideState_mode == GUIDE_NOOP ->
     printf("mode GUIDE_NOOP\n");
     skip;
 fi
}
```

Listing 5: Cognition logic for evaluating triggers.

```
bool events[3] = {false, false, false};

inline RunEventMonitor(elem) {
 if
  :: events[elem] -> bool avail = false;
     if
      :: cogState_activeHandlers[elem] ->
         avail = true;
      :: else -> skip;
     fi

     if
      :: !avail -> execState[elem] = NOOP;
         if
          :: !IsEmpty_activeHandlers ->
             byte currHandler;
             GetTopPriorityActiveHandler
              (currHandler);
             cogState_activeHandlers[elem] =
              true;
             byte topHandler;
             GetTopPriorityActiveHandler
              (topHandler);
             if
              :: currHandler == topHandler ->
                 skip;
              :: else ->
                 execState[currHandler] =
                  TERMINATE;
                 RunEvent(currHandler, _);
                 execState[currHandler] =
                  DONE;
             fi
          :: else ->
             cogState_activeHandlers[elem] =
              true;
         fi
      :: else -> skip;
     fi
   :: else -> skip;
 fi
}

inline RunEventMonitors()  {
  printf("RunEventMonitors()\n");

  bool TakeoffTrigger =
   (cogState_missionStart == PRE_MISSION);
  events[TAKEOFF_PHASE_HANDLER] =
   TakeoffTrigger;
  RunEventMonitor(TAKEOFF_PHASE_HANDLER);

  bool NominalDepartureTrigger =
   (cogState_missionStart == FLIGHT &&
    !cogState_nominalPlanEngaged);
  events[ENGAGE_NOMINAL_PLAN] =
   NominalDepartureTrigger;
  RunEventMonitor(ENGAGE_NOMINAL_PLAN);

  bool PrimaryPlanCompletionTrigger =
   (cogState_nextWpId >= activePlanSize);
  events[LAND_PHASE_HANDLER] =
   PrimaryPlanCompletionTrigger;
  RunEventMonitor(LAND_PHASE_HANDLER);
}
```

Listing 6: Handlers in Cognition and a Cognition method used by a handler to set the flight plan.

```
mtype:execState_e
 execState[3] =  {NOOP, NOOP, NOOP};

inline SetGuidanceFlightPlan(wp_index) {
 command_t cmd;
 cmd.commandType = FP_CHANGE;
 cmd.wpIndex = wp_index;
 cogState_cognitionCommands!cmd;
}

inline TakeoffPhaseHandler_Initialize(ret) {
 printf("TakeoffPhaseHandler_Init\n");
 command_t cmd;
 cmd.commandType = TAKEOFF_COMMAND;
 cogState_cognitionCommands!cmd;
 cogState_missionStart = LAUNCH;
 cogState_takeoffState = TAKEOFF_INPROGRESS;
 ret = SUCCESS;
}

inline TakeoffPhaseHandler_Execute(ret) {
 printf("TakeoffPhaseHandler_Execute\n");
 if
   :: cogState_takeoffState ==
       TAKEOFF_COMPLETE -> ret = SUCCESS;
   :: else -> ret = INPROGRESS;
 fi
}

inline TakeoffPhaseHandler_Terminate(ret) {
 printf("TakeoffPhaseHandler_Terminate\n");
 if
  :: cogState_takeoffState ==
      TAKEOFF_COMPLETE ->
     cogState_nextWpId = 1;
     cogState_missionStart = FLIGHT;
  :: else -> skip;
 fi
 ret = SUCCESS;
}

byte setGuidanceFlightPlanCounter = 0;
inline EngageNominalPlan_Initialize(ret) {
  printf("EngageNominalPlan_Init()\n");
  SetGuidanceFlightPlan(cogState_nextWpId);
  setGuidanceFlightPlanCounter++;
  cogState_icReady = true;
  cogState_nominalPlanEngaged = true;
  ret = SUCCESS;
}

inline EngageNominalPlan_Execute(ret) {
  printf("EngageNominalPlan_Execute()\n");
  ret = INPROGRESS;
}

inline EngageNominalPlan_Terminate(ret) {
  printf("EngageNominalPlan_Terminate()\n");
  cogState_nominalPlanEngaged = false;
  ret = SUCCESS;
}

inline LandPhaseHandler_Execute(ret) {
  printf("LandPhaseHandler_Execute()\n");
  command_t cmd;
  cmd.commandType = LAND_COMMAND;
  cogState_cognitionCommands!cmd;
  cogState_missionStart = LANDING;
  ret = SUCCESS;
}
```

Listing 7: Cognition method to run top priority active handler and engage the EventManager.

```
inline RunEventHandlers() {
 printf("RunEventHandlers()\n");
 if
  :: IsEmpty_activeHandlers ->
     goto Ret_REH;
  :: else -> skip;
 fi

 byte topHandler;
 GetTopPriorityActiveHandler(topHandler);

 isDone = false;

 if
  :: execState[topHandler] == NOOP ->
     if
      :: events[topHandler] ->
         execState[topHandler] =
          INITIALIZE;
         // Event Manager state machine
         RunEvent(topHandler, isDone);
      :: else ->
         cogState_activeHandlers
          [topHandler] = false;
     fi
  :: else -> RunEvent(topHandler, isDone);
 fi

 if
  :: isDone ->
     cogState_activeHandlers[topHandler] =
      false;
  :: else -> skip;
 fi

Ret_REH:  skip;
}
```

Listing 7 shows portions of the model that capture the Event Manager's method for running the top priority handler. If the handler's state is *NOOP*, then it first checks whether the corresponding trigger is still true. If it is, then it sets its state to *INITIALIZE* and calls *RunEvent()* to process it according to the Event Manager's state machine logic. If it is not, then it records the handler as inactive in Cognition's state. If the handler's state is not *NOOP*, then it simply calls *RunEvent()* to process it. If the result of processing it is that the handler is done, it records the handler as now inactive in Cognition's state.

4.2 Verification of ICAROUS model in Spin

The listings in Section 4.1 show the final version of the ICAROUS Spin model. However, part of the motivation for this work is that originally, the handler for engaging the nominal flight plan was not behaving as expected. Modeling, simulating, and verifying ICAROUS in Spin helped uncover the reasons for unexpected behaviors and explore potential fixes. LTL specifications used for verification are given in Listing 8. Portions of the model that changed over the course of this process are colored light blue in Listings 1, 5, and 6, and the reasons for the changes are explained here.

Listing 8: LTL specifications for the Spin model of ICAROUS.

```
ltl missionStartFlight { <> (cogState_missionStart == FLIGHT) }
ltl setGuidanceFlightPlanOnce { <> (setGuidanceFlightPlanCounter == 1) &&
                                [] (setGuidanceFlightPlanCounter <= 1) }
ltl engagedOnThenOff { <> (cogState_nominalPlanEngaged && <> (!cogState_nominalPlanEngaged)) }
ltl guidanceReachesFlightPlanEnd { <> (guideState_nextWpId == activePlanSize)  }
ltl stackLanding { (!stackState_land U stackState_nextWP1 == activePlanSize) &&
                   <> (stackState_land) }
ltl landSteps { <> (landSteps > 0) }
ltl lastWaypointsReachedInOrder { <> (guideState_nextWpId == 4  &&
                                      <> (guideState_nextWpId == 5)) }
ltl lastWaypointsNotReachedOutOfOrder { guideState_nextWpId != 5 U guideState_nextWpId == 4 }
```

Simulation of the first version of the model did not terminate. Note that the model contains simple print statements to output basic information, such as when certain inline procedures are called and what the values of triggers are when they are evaluated. By default, these are printed during a Spin simulation. Reviewing simulation outputs showed that after the takeoff phase handler terminated, all triggers remained false. Indeed, the verification mode of Spin can be used to check for non-progress cycles, and it found one in which after the takeoff phase handler terminated, the model infinitely cycled through running the Guidance and Cognition modules, yet all trigger values remained false. Inspection of the model showed that the trigger for the engage nominal plan handler expected the *missionStart* attribute of Cognition to be set to *FLIGHT*, but this never occurred. Using Spin to verify this expected property, encoded as LTL specification *missionStartFlight*, returned the same counterexample. A statement to set this variable in the takeoff phase handler's *Terminate()* method was therefore added, leading to the second version of the model.

Simulation of the second version of the model did terminate, and verification of LTL specification *missionStartFlight* was successful. However, inspection of the simulation output showed an unexpected behavior. The engage nominal flight handler cycled several times through its *Initialize()*, *Execute()*, and *Terminate()* methods. The logic now in this handler's *Initialize()* method was originally in its *Execute()* method instead, except there was no Cognition attribute *nominalPlanEngaged* to track whether the nominal plan is engaged, no counter *setGuidanceFlightPlanCounter* to track the number of times a guidance flight plan is set, and the *Execute()* method returned value *SUCCESS* instead of *INPROGRESS*. Given that the *Execute()* method called *SetGuidanceFlightPlan()*, the fact that it was called repeatedly was undesirable, since the nominal flight plan only needs to be set once. Attempting to verify LTL specification *setGuidanceFlightPlanOnce*, which specifies that the counter should reach a value of one but not exceed it, resulted in a counterexample. This counterexample showed that the trigger for the engage nominal plan handler remained true, and since each of the handler's methods returned *SUCCESS*, the cycling behavior followed from the logic of the Event Manager state machine encoded in procedure *RunEvent()* when *RunEventHandlers()* was called, until the higher priority trigger for the land phase handler became true. The call to *SetGuidanceFlightPlan()* and the

corresponding counter was moved to the handler's *Initialize()* method so that it would only get called once, along with the statement to set Cognition's *icReady* flag to true. The return value of the *Execute()* method was changed to *INPROGRESS*, so that the handler would not automatically transition from its *EXECUTE* to *TERMINATE* state. A Cognition state variable to track whether the nominal plan is engaged was also added to the model. It is initialized to false, set to true in the handler's *Initialize()* method, and set to false in the handler's *Terminate()* method. These changes led to the third version of the model.

Simulation of the third version of the model showed that the engage nominal plan handler no longer cycled between its three methods. LTL specifications *missionStartFlight* and *setGuidanceFlightPlanOnce* were verified to hold. However, the simulation output showed that its *Terminate()* method was never called. LTL specification *engagedOnThenOff* encodes the property that eventually Cognition's attribute *nominalPlanEngaged* becomes true and then later eventually becomes false, which should happen in the *Terminate()* method. Verification of this property resulted in a counterexample. This counterexample showed that when the higher priority land phase handler became active while the engage nominal plan handler was still active, the Event Manager set the engage nominal plan handler's state to *DONE*, bypassing execution of its *Terminate()* method. A change to *RunEventMonitor()* was made to first set the interrupted handler's state to *TERMINATE*, run the handler, and then set its state to *DONE*. This led to the fourth and final version of the model, for which the desired properties were verified to hold.

Verification of additional LTL specifications can be used to continue to build confidence in the model. For example, the LTL specification *guidanceReachesFlightPlanEnd* was used to check that Guidance eventually sets its next waypoint index to the length of the flight plan, which is one index past the last waypoint (due to zero-based indexing). LTL specification *stackLanding* was used to check that eventually Stack records that the next waypoint index is the length of the flight plan, that it does not register landing before that, but it does eventually register it. LTL specification *landSteps* was used to check that a non-zero number of steps are taken during landing. LTL specifications *lastWaypointsReachedInOrder* and *lastWaypointsNotReachedOutOfOrder* were used to check that waypoint 4 is eventually reached and waypoint 5 is later reached, but waypoint 5 is not reached before waypoint 4. Note that longer versions of these two specifications that include all waypoints cannot be fully analyzed in Spin with the default settings. This is because, in the worst case, the time required for verification grows exponentially with number of temporal operators in an LTL formula [5]. Even increasing the search depth and available memory significantly did not enable Spin to fully explore the model with respect to those specifications. State explosion is a well-known drawback of model checking, and in future iterations of this work, different techniques to reduce the number of states will be explored.

5. Discussion and Conclusions

In this paper, the use of model checking to model and verify the ICAROUS architecture for building safety-centric autonomous unmanned aircraft was explored. Key ICAROUS capabilities were modeled in Promela, the language of the Spin model checker. Then an iterative verification process was performed in which Spin was first used to simulate the model, then it was used to verify through model checking whether the model satisfied LTL specifications expressing desired behaviors of the modeled system. Both simulation results and counterexamples produced through model checking were used to find and fix errors in the model. As demonstrated in this application, simulation provides a fast way to find relatively straightforward errors, and model checking provides a way to find more complex errors and ultimately prove that a model satisfies a set of given specifications.

The original motivation for this work was that the handler for engaging the nominal flight plan after takeoff in ICAROUS was not triggering as expected. Instead, code was added to run this handler manually after takeoff. The iterative process of simulation and model checking discussed in this paper helped identify the issue that was preventing the trigger from firing, then refine a candidate solution that would not have undesirable side-effects. Changes to the model were implemented in an experimental version of the code, and the code for manually running the handler after takeoff was

removed. The ICAROUS development team has a test suite containing 44 continuous integration tests. The experimental version of the code passed all but one of these tests. The test that failed is one of nine tests that check interactions with the Traffic Monitor, which was not included in this model. Unfortunately, when a test does not pass, it is not always easy to understand the underlying cause. There are several reasons for this. One is that the system is a mix of Python and C++ code, which makes it hard to run a debugger and step through a particular execution of the system. Another is that the full system has a very large state composed of the state of many individual components whose interactions and effects on each other can be difficult to reason about. For systems such as this, failures due to erroneous interactions between components are often some of the most difficult to identify and understand. A model of the system that targets component interactions and abstracts out other details makes it easier to find these types of errors. Model checking provides feedback in the form of simulation paths and counterexamples that are generally easier to understand, though it does take additional time to develop the model. Future work will continue to expand this model of ICAROUS to include additional components such as the Traffic Monitor, with the goal of finding a solution that both allows the handler for engaging the nominal flight plan to trigger as expected and for all the continuous integration tests to pass.

Other future work will focus on modeling ICAROUS as it runs with cFS. In this paper, Pycarous and the Autonomy Stack are used to coordinate interactions between ICAROUS components, and there is only a single thread of execution in the code that is driven by Pycarous. This leads to a model that is not that complex in the sense that it only consists of a single process with very little non-determinism in how it executes; the only non-determinism is in the number of time steps it takes to reach each waypoint and to land. In fact, this model does not have many features that Spin is geared toward analyzing. In cFS however, ICAROUS components run in asynchronous threads and communicate over a bus using a publisher-subscriber messaging pattern. Non-deterministic interleaving between processes can lead to different behaviors, some of which occasionally cause continuous integration tests to fail. Since the Spin model checker is well-suited to this type of problem, it was chosen for this preliminary modeling exercise, which will serve as a starting point for identifying possible errors in ICAROUS when it runs with cFS and exploring potential solutions.

6. Contact Author Email Address

The primary author's email address is laura.r.humphrey@nasa.gov.

7. Copyright Statement

References

[1] L Humphrey. Example applications of formal methods to aerospace and autonomous systems. In *IEEE International Conference on Assured Autonomy (ICAA 2023)*, pages 67–75, 2023.

[2] G J Holzmann. Mars code. *Communications of the ACM*, 57(2):64–73, 2014.

[3] K Havelund, M Lowry, and J Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

[4] A Cimatti, E Clarke, E Giunchiglia, F Giunchiglia, M Pistore, M Roveri, R Sebastiani, and A Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification (CAV 2002)*, pages 241–268, 2002.

[5] G J Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, 1997.

[6] S Eker, J Meseguer, and A Sridharanarayanan. The Maude LTL model checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2004.

[7] A Champion, A Mebsout, C Sticksel, and C Tinelli. The Kind 2 model checker. In *International Conference on Computer Aided Verification (CAV 2016)*, pages 510–517, 2016.

[8] M Kwiatkowska, G Norman, and D Parker. PRISM 2.0: A tool for probabilistic model checking. In *IEEE First International Conference on Quantitative Evaluation of Systems (QEST 2004)*, pages 322–323, 2004.

[9] A David, K G Larsen, A Legay, M Mikučionis, and D B Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17:397–415, 2015.

[10] S Balachandran, C Muñoz, M Consiglio, M Feliú, and A Patel. Independent configurable architecture for reliable operation of unmanned systems with distributed on-board services. In *IEEE/AIAA 37th Digital Avionics Systems Conference (DASC 2018)*, 2018.

[11] E Pastor, C Barrado, P Royo, J Lopez, and E Santamaria. An open architecture for the integration of UAV civil applications. In T M Lam, editor, *Aerial Vehicles*, chapter 24. IntechOpen, Rijeka, 2009.

[12] C Muñoz, A Narkawicz, G Hagen, J Upchurch, A Dutle, and M Consiglio. DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In *IEEE/AIAA 34th Digital Avionics Systems Conference (DASC 2015)*, 2015.

[13] M M Moscato, L Titolo, M A Feliú, and C A Muñoz. Provably correct floating-point implementation of a point-in-polygon algorithm. In M H ter Beek, A Mclver, and J N Oliveira, editors, *Formal Methods – The Next 30 Years*, volume 1180 of *Lecture Notes in Computer Science*, pages 21–37. Springer International Publishing, 2019.

[14] A Peters, B Duffy, S Balachandran, M Consiglio, and C Muñoz. SIRIUS: Simulation Infrastructure for Research on Interoperating Unmanned Systems. In *IEEE/AIAA 40th Digital Avionics Systems Conference (DASC 2021)*, 2021.

[15] S Merz. Model checking: A tutorial overview. *Summer School on Modeling and Verification of Parallel Processes*, pages 3–38, 2000.