Run-Time Assurance for Learning-Based Aircraft Taxiing

Darren Cofer, Isaac Amundson,
Ramachandra Sattigeri,
Arjun Passi, Christopher Boggs
Collins Aerospace
first.last@collins.com

Eric Smith,
Limei Gilham
Kestrel Institute
{eric.smith, gilham}@kestrel.edu

Taejoon Byun,
Sanjai Rayadurgam
University of Minnesota
{taejoon, rsanjai}@umn.edu

***Abstract*—Aircraft systems that include *learning-enabled components (LECs)* and their software implementations are not amenable to verification and certification using current methods. We have produced a demonstration of a run-time assurance architecture based on a neural network aircraft taxiing application that shows how several advanced technologies could be used to ensure safe operation.**

***Index Terms*—machine learning, run-time assurance**

I. Introduction

Significant advances are being made in the development of autonomous systems that employ learning and adaptation algorithms. It is therefore inevitable that *learning-enabled components* (LEC) will begin to find their way into safety-critical applications, including manned and unmanned vehicles. However, the technologies being applied — machine learning, deep neural networks, probabilistic languages — are not amenable to verification using traditional methods. This essentially precludes use of these technologies in many safety-critical aerospace applications. Our team is developing technologies to overcome these limitations, thus expanding opportunities for autonomous systems to be safely deployed in aerospace applications.

Aircraft systems have legal requirements for airworthiness certification that present barriers to the use of LECs. In a typical LEC, much of the complexity and design information resides in its training data rather than in the actual code produced. For example, one of the key principles of avionics software certification (covered in DO-178C [20]) is the use of requirements-based testing along with structural coverage metrics. These activities not only demonstrate compliance with functional requirements, but are intended to show the absence of unintended functionality. However, it can be difficult to precisely state requirements for LECs, especially those implementing perception functions. Even when requirements are available, it is usually not possible to associate any particular lines of code with a specific requirement. Furthermore, complete structural coverage can be achieved for a typical neural network with a single test case, but this provides almost no confidence in its correctness. Showing that a component or system is correct *and* does not do harm because of behaviors that were unintended by designers or unexpected by operators is a critical aspect of the certification process. Additional details regarding certification challenges posed by learning-based systems can be found in [3].

Since it is difficult to demonstrate assurance by examining the LEC itself (as is assumed by existing certification processes) other approaches are needed. In this paper we report on the use of a *run-time assurance* architecture based on the ASTM F3269-17 standard for bounded behavior of complex systems [2], also known as a *simplex* architecture [21]. The standard provides guidance for mitigating unintended functionality (such as may be present in a LEC) through the use of run-time monitors. When a violation of system safety properties or an unsafe LEC output is detected, the architecture switches to a verified backup controller to continue safe operation. The main idea is that system performance is provided by the complex system or LEC while system safety is guaranteed by high-assurance components (though with lower performance). Our implementation of the standard includes:

- System architecture modeled using the *Architecture Analysis and Design Language* (AADL) [11]
- Formal verification of system behaviors using the *Assume Guarantee Reasoning Environment* (AGREE) [22]
- Architecture-based assurance case for showing correct implementation using Resolute [10]
- Diverse run-time monitors for system safety, integrity, and availability
- Synthesis from formal specifications with proof of correctness for critical high-assurance components

The purpose of this paper is to show the effectiveness of a run-time assurance architecture for bounding the behavior of an autonomous system to maintain its safety requirements. In this example, surface movement of a general aviation class aircraft is controlled during taxi based on a position estimate computed by an LEC. Our work illustrates the general effectiveness of the run-time assurance approach and demonstrates some of the tools and methods that can be applied in a real system. However, each specific application will require different monitors and backup safety functions, depending on requirements and variables that can be monitored. We discuss some of the challenges and limitations in Section V.

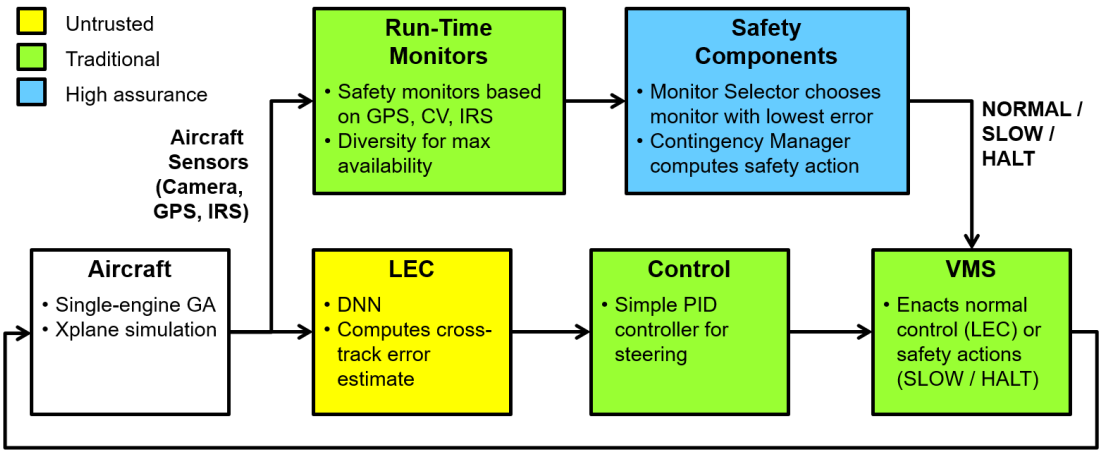

Fig. 1: TaxiNet Demonstration System with Run-Time Assurance Architecture

II. Demonstration

The "TaxiNet" demonstration system is shown in Figure 1. The bottom row of boxes in the figure show the baseline system, consisting of the aircraft (or simulation), the guidance LEC, a controller for steering the aircraft, and the Vehicle Management System (VMS) which manages actuators on the aircraft and integrates other autonomy functionality. The LEC is implemented as a deep neural network (DNN) trained to estimate the cross-track error (CTE) of the aircraft (position left or right of the runway centerline) based on images from a forward-looking camera on the aircraft. Since the images are 360x200 pixels, the resulting LEC is larger than can be analyzed by current formal methods tools for DNNs, such as Marabou [13].

The system can run on a small single-engine aircraft using the LEC for autonomous taxiing. An equivalent high-fidelity simulation environment is also available for testing and demonstrations. This is based on a detailed 6 degree-of-freedom physics simulation and the Xplane program for visualization.

Six different LECs trained in various lighting and weather conditions were created for testing. The training data for the six LECs are as follows:

1) All environmental conditions without data augmentation effects (blurring, lens effects, etc.)
2) All environmental condition with data augmentation
3) Conditions presenting in the earlier morning
4) Conditions presenting in the late afternoon
5) Clear day conditions
6) Overcast day conditions

Thus, LEC 2 is the most general and robust, while LECs 3-6 have specific limitations. This allows faulty behaviors to be simulated by operating an LEC in conditions other than those for which it was trained.

This autonomy framework was developed by the Boeing Research and Technology (BR&T) organization and is being used as a demonstration platform in DARPA's Assured Autonomy program [8].

The run-time assurance architecture adds components in the top row of Figure 1. This includes four different run-time monitors (three for system safety, one for LEC confidence), a Monitor Selector for choosing which monitor to use at any time, and a Contingency Manager that determines when intervention is needed to maintain safety and what action should be taken. In this example, the safety actions available (via the VMS) are to reduce the commanded aircraft speed or to use the brakes to halt the aircraft.

The goal of the run-time assurance architecture is to ensure that the LEC does not result in violation of aircraft safety requirements. While the LEC is responsible for performance (tracking the center line), the safety requirements for this application are to ensure that 1) the aircraft does not deviate too far from the center line and leave the runway and 2) unnecessary stopping on the runway is minimized. In the baseline system, the LEC cannot be verified using traditional means and is considered the complex or untrusted component in the run-time assurance architecture. The run-time safety monitors, PID controller, and the VMS are either based on existing verified algorithms or have been developed using traditional methods compatible with DO-178C. The Monitor Selector and Contingency Manager are high-assurance components that are synthesized and verified using formal methods.

III. Approach

The three elements of our run-time assurance approach are the architecture itself, the run-time monitors, and the safety components that manage selection of behaviors.

A. Architecture

The assurance architecture has been modeled using AADL. AADL is targeted at distributed, real-time-embedded systems and provides sufficiently rigorous semantics to support formal analysis of system safety properties. We use an extension of AADL called AGREE to annotate the model with formal assume-guarantee contracts for components and subsystems.

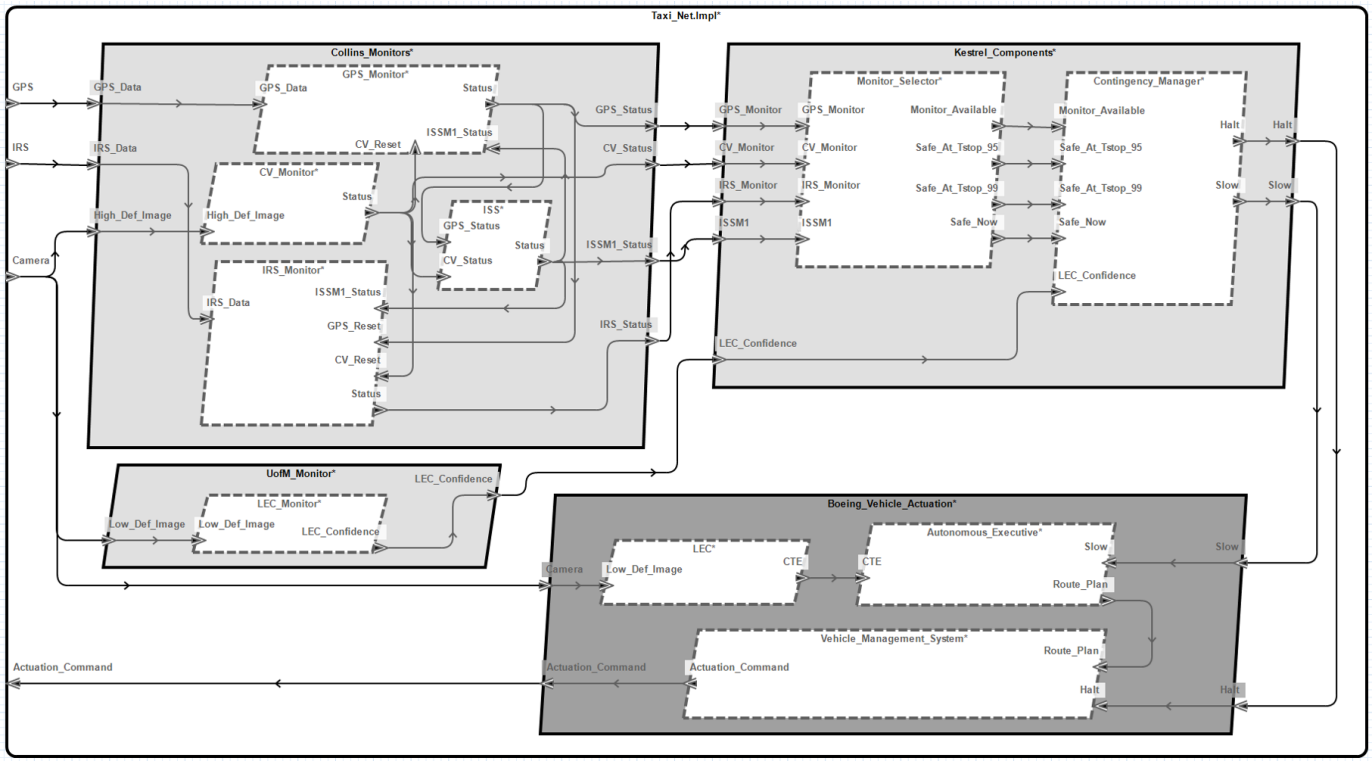

Fig. 2: AADL Model of Run-Time Assurance Architecture

AGREE uses the JKind k-induction model checker [12] to verify the top-level contracts of the system using a compositional approach [22].

AADL includes both textual and graphical syntax. A diagram of the run-time assurance architecture for the demonstration system is shown in Figure 2. The diagram shows the baseline aircraft systems on the bottom right, the run-time monitors on the top left, and the high-assurance monitor selection and contingency manager components on the top right. AGREE contracts for each component are specified in the textual representation of the model.

Verification of the safety properties in our demonstration system relies on the correctness of the monitors and other safety components, and establishes system safety for each state of the architecture. Figure 3 provides an example of one of the system safety properties specified in AGREE. This property states that if none of the run-time monitors are available, then the aircraft will be commanded to halt. The AGREE model checker is able to verify that this property holds under all conditions.

The AADL model also includes an *assurance case* embedded in the architecture using the Resolute language [10]. Resolute assurance cases are linked to architectural components and formal evidence produced by other tools, and can be continuously re-evaluated during system development to check for errors. We have used Resolute to produce an assurance case showing that the run-time assurance architecture has been used correctly and has not been compromised by other elements of

```
annex agree {**
    -- HALT will be commanded if no monitors are available
    property no_mon_avail = false ->
        pre(not (GPS_Monitor.Monitor_Available or
                 CV_Monitor.Monitor_Available or
                 IRS_Monitor.Monitor_Available
        ));
    guarantee "HALT" : no_mon_avail => Halt;
**};
```

Fig. 3: Example safety property specified in AGREE

the system design. It also addresses the goal of minimizing unnecessary stopping through the use of independent monitors to maximize availability.

The Resolute assurance case for part of the demonstration system is shown in Figure 4. This portion of the assurance case focuses on evidence derived from the run-time assurance architecture. Major branches of the argument can be seen in the figure, corresponding to the use of multiple diverse monitors, formal synthesis of high-assurance components, and selection of contingency actions to minimize unnecessary stopping.

In other projects we have demonstrated how the implementation can be built from the verified AADL model for execution on the formally verified seL4 kernel [7]. While outside the scope of the current project, the isolation provided by seL4 adds assurance that a malfunctioning LEC does not have any computational side effects (memory or execution time) on the rest of the system.

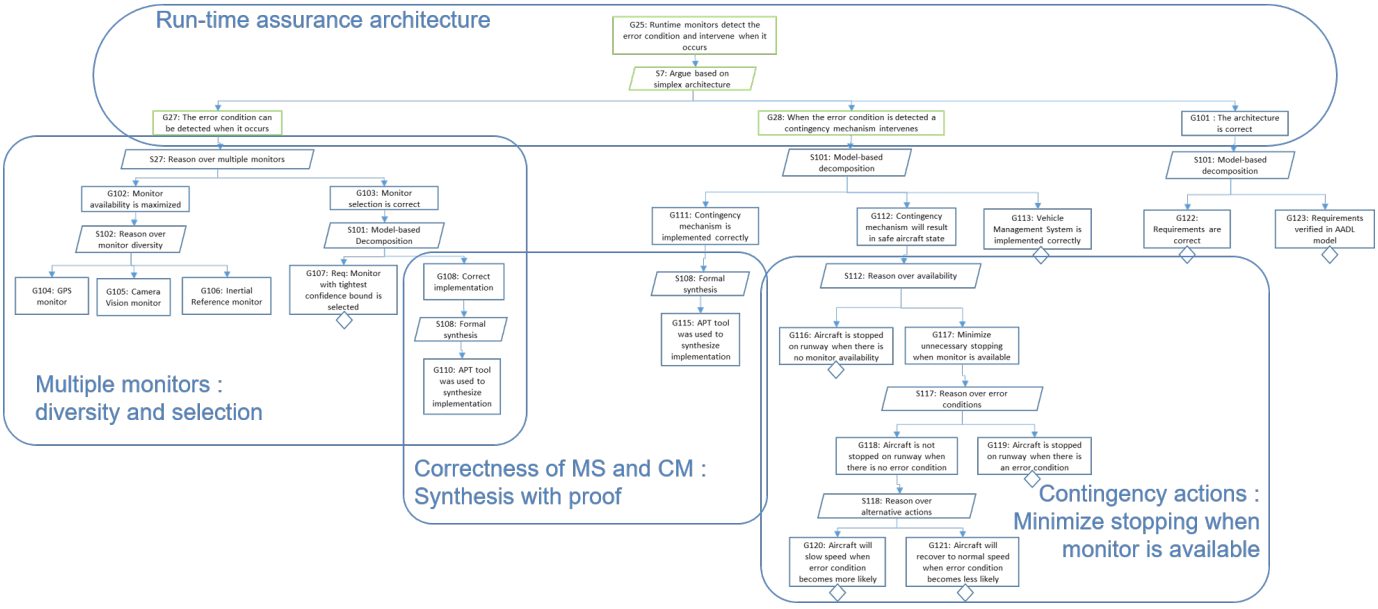

Fig. 4: Assurance Cases for AADL model generated using Resolute

B. Run-Time Monitors

The assurance architecture includes three independent monitors of system safety and one monitor that assesses confidence in the current operation of the LEC.

The first monitor uses Global Positioning System (GPS) signals to compute the aircraft position relative to a virtual runway centerline. Specifically, high precision carrier phase (CP) signals from a GPS receiver [19] are differenced in time to generate very accurate GPS velocity measurements[1].The initial conditions for the system require that the aircraft be positioned on the runway centerline. That is, its cross-track deviation from the centerline is small and it is aligned with the centerline.

Prior to beginning to taxi, the aircraft position on the runway is latched. As the aircraft moves down the runway, the high accuracy GPS velocity measurements are integrated over time to compute the position change of the aircraft relative to the starting location. The accuracy of this relative position solution is on the order of centimeters, with an error that grows at the rate of few mm/second driven primarily by satellite clock rate drift. The relative position is computed in the local-level coordinate plane (North-East-Down axes). The relative position in the local-level coordinate frame is transformed to a runway coordinate frame to compute translation along and lateral to the runway centerline (CTE). Runway database information in the form of runway start and end coordinates and runway width, are assumed to be available with high accuracy and integrity, to perform this position transformation.

In addition to the estimate of CTE, the GPS monitor also provides a high confidence error bound for the estimate. This bound captures the effect of all error sources including the GPS sensor errors, ability of the pilot or autopilot to follow the centerline, and initial positioning errors. This approach executes with low overhead with a solution update rate of 1 Hz. Software development was based on standard verification and validation techniques. Note that the GPS monitor may become unavailable due to its growing error bound or in case of poor satellite geometry leading to a solution with large errors (errors that are the magnitude of the safety thresholds defined for the centerline tracking problem).

The second monitor processes camera images and detects the runway centerline using traditional computer vision (CV) algorithms. It can be used if GPS is unavailable or monitor error becomes too high. The CV monitor approach is based on the highly structured and regulated nature of runway markings (e.g., repeating dashed line pattern of known dimensions). It tracks these features in the image sequence as the aircraft taxis down the runway. The algorithm receives as inputs the high-resolution images from the same camera used by the LEC. In also receives the aircraft attitude from the onboard navigation system at the instant of image acquisition and the camera calibration parameters. The output of this algorithm is a secondary estimate of CTE with an error bound.

The CV monitor output is also used to periodically reset the GPS monitor solution when the CV monitor error bound is smaller than the corresponding error bound of the GPS monitor. This stops the growth in the error of the GPS monitor solution and allows the GPS monitor to become available for longer periods of time.

The centerline detection algorithm is outlined below:

1) Receive algorithm inputs: image, attitude from onboard navigation system, camera position and orientation, and calibrated camera model.

[1]Some GPS receivers don't output CP measurements, but they can output delta-range measurements which are internally computed inside the receiver as time-differenced CP signals.

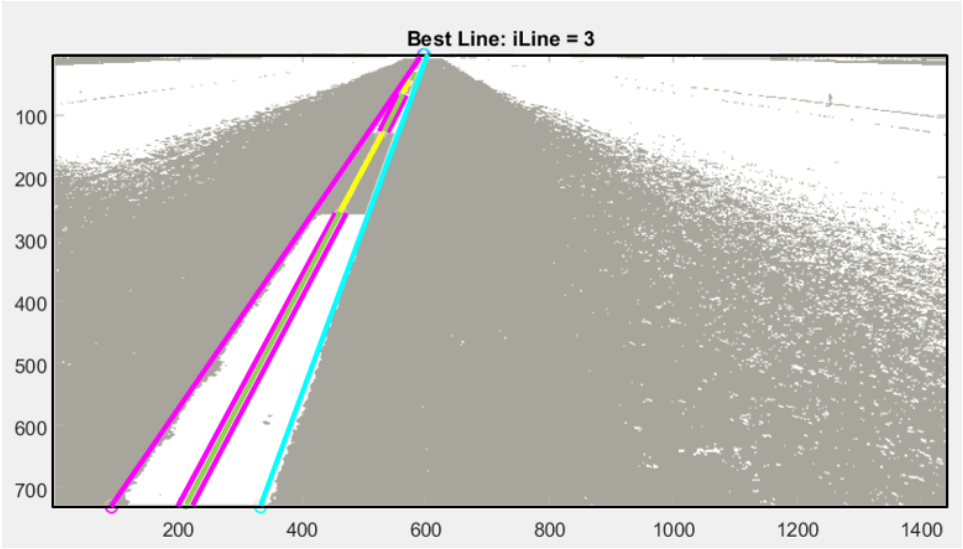

Fig. 5: Illustration of Centerline Detection by CV Monitor

2) Compute transform from aircraft heading frame of reference to camera frame of reference.
3) Extract candidate line segments from image using Hough transform.
4) Test each line segment as a candidate edge of the dashed runway centerline:
    a) Extrapolate candidate line segment to edges of image and find its endpoints
    b) Inverse-project the endpoints to obtain 3D points in the aircraft heading frame.
    c) Construct candidate 3D centerline dash-gap square wave patterns corresponding to different phases and project into the camera frame.
    d) Compute the correlation between detected lines and projected square wave patterns for each phase.
    e) Select the best phase and the best line to use for the aircraft position estimate, subject to detection thresholds.
    f) The detected line is transformed to the aircraft heading frame to compute CTE.

An illustration of the centerline detection result is shown in Figure 5. There are periods of time when the CV solution is not available due to the centerline being obscured by skid marks or the centerline pattern not being detected reliably due to illumination changes or aircraft heading. The CV monitor makes use of existing algorithms, but requires fairly large computing overhead. Running at 1 Hz, it makes full use of two CPU cores.

The third monitor uses the high-integrity inertial reference system (IRS) measurements to compute aircraft position. It provides coverage when both the GPS and CV monitors are unavailable. This monitor uses acceleration and attitude data from the IRS to propagate the most recent CTE estimate and error bound from the GPS or CV monitors. The IRS data also ensures continuity of the monitoring solution at high rate (50 Hz) between consecutive primary sensor updates or in the presence of measurement drop-outs. It also makes use of existing code and traditional verification techniques, and executes with low overhead.

The three safety monitors above are implemented using sensors that are typically installed on aircraft (GPS, IRS, camera). The monitors are complementary in their nature. If one of the primary monitors is lost, perhaps due to large GPS errors or CV failing to detect the centerline due to skid marks or low visibility, they work together to enhance overall monitoring availability. This minimizes unnecessary interventions by the run-time assurance architecture in nominal (fault-free) conditions and helps ensure correct intervention when the LEC outputs steer the aircraft away from the centerline.

The final monitor is used to determine if the LEC is operating in a region of competence by assessing the novelty of the input image relative to the training data. The rationale behind this approach is that the LEC being monitored may produce incorrect output more frequently for novel (out-of-distribution) inputs since it was not trained on them. However, the underlying probability distribution of the high-dimensional data—such as images—is unknown, making it impossible to compute the exact likelihood of an input given the training data. In this work, we adpoted a learning-enabled approach called Variational Autoencoder (VAE) [9] which can estimate this underlying distribution, and leverage it for detecting the novelty of the input.

VAE is an unspupervised learning technique for obtaining a compact representation of the training data along with an encoder and a decoder which map the input data to and from the latent representation space. A typical VAE consists of a pair of neural networks—1) a probabilistic encoder which compresses an input data to a latent code distribution, and 2) a decoder which can reconstruct an input image from a given latent code. The two networks are connected through a bottleneck which

forces the necessary information for the generative model to be compressed and passed over to the decoder. During training, these two networks are jointly optimized towards two goals—1) minimize the difference between the original input and the reconstructed (encoded and then decoded) input, and 2) regularize the distribution of the latent code towards a pre-imposed prior distribution so that the underlying distribution of the input data is modeled as a tractable distribution. The outcome of this optimization is a latent space from which new inputs can be synthesized along the training data distribution.

The intuition behind utilizing VAE for novelty detection is to either harness the limited capability of the decoder—as it can only generate inputs that are similar to the training data—or use the distribution of the latent space for determining the typicality of an encoding [1], [4]. Several approaches have been proposed [6], [18], and their performance may vary depending on the task that LEC performs and the type of novelty—adversarial vs. non-adversarial—one aims to detect. In this work, we adopted a simple reconstruction-based novelty detection which computes the score based on the pixel-wise difference between the input image and the reconstructed image.

One may question the assurance of using yet another LEC for monitoring an LEC, as the VAE-based monitor is not a trusted component. Indeed, they may seem to suffer from the same mode of failure since both are trained on the same data. However, VAE monitor is capable of detecting an input that drifts away from the *known* data, and it works especially well when the novel input is visually distinct. The LEC being monitored, on the other hand, does not have such a capability of signaling the unknowns and keeps on producing incorrect outputs silently even when it is incapable of handling such inputs.

Since this monitor is not a trusted component, it was not used to enforce system safety, but rather as an additional check on LEC performance. We used it to allow the Contingency Manager to recover from a transient SLOW or HALT action if the current safety monitor output returns to normal and the LEC appears to be in its region of competence. The monitor can be relatively expensive when running on a CPU alone because of the inference time required for the neural network, with one encoding and decoding per input. However, much performance can be gained by using a GPU. It is also much less computationally intensive than other alternatives such as estimating uncertainty directly from the LEC using Monte-Carlo simulation [5].

C. Safety Components

The Monitor Selector and the Contingency Manager are critical for safe operation (single instance, no backup) and so have been implemented as high-assurance components using formal synthesis. They provide inputs to the VMS that determine the control action to be taken to guarantee system safety.

The Monitor Selector must choose which of the three safety monitors should be used at each time step. If GPS and CV are both available, it chooses the one with the smallest error bound (subject to minimum switching time). If neither GPS nor CV is available, it uses the IRS monitor.

The Contingency Manager determines whether control should be based on LEC outputs (NORMAL) or one of the contingency actions (SLOW or HALT). Roughly speaking, SLOW speed is selected if the current CTE exceeds the 'slow' threshold or the predicted stopping position based on the current speed is too close to the runway edge. HALT is selected if the current CTE exceeds 'halt' threshold or the predicted stopping position is too close to runway edge. Recovery from SLOW or HALT is allowed if the LEC confidence monitor output is above its threshold and a specified time limit has not been exceeded.

Both the Monitor Selector and the Contingency Manager are synthesized from formal specifications with proofs of correctness of their core functionality. The synthesis is done in the ACL2 theorem prover using the Automated Program Transformations (APT) toolkit. ACL2 (A Computational Logic for Applicative Common Lisp) is a software system consisting of a logic, a programming language, and an automatic theorem prover [17]. The APT toolkit is a software synthesis system built on top of ACL2 [14]. Specifications written in ACL2 are often high-level and should formalize the requirements for the software components. They may be unexecutable, requiring refinement to executable versions. The APT toolkit includes automated transformations which can be used to refine high-level specifications to low-level, executable, efficient, and provably correct implementations. Each APT transformation represents a single design choice, such as the selection of a divide-and-conquer algorithm or the application of an optimization such as incrementalization. Each application of an APT transformation to a specification generates a new (lower-level) specification and a proof of correctness. Applying a sequence of APT transformations to a formal specification results in ACL2 code that provably satisfies the requirements of the software component.

After analyzing the functional requirements for the Monitor Selector and the Contingency Manager, we chose to specify these components in terms of *tables*. A *table* specifies the behavior of a component declaratively as a set of cases corresponding to the columns of the table. Each case specifies the values of input expressions to match and the values of outputs to set following a match.

For example, Figure 6a shows a small stateful table specifying how the Monitor Selector stabilizes the `ISSM1` input signal (a flag indicating that whether the GPS monitor has a smaller error bound than the CV monitor). This prevents rapid switching between the GPS monitor (`M1`) and the CV monitor (`M2`). In this table, each case contains values of three input expressions: `Current State`, `ISSM1`, and `timestamp` $\geq$ `EarliestSwitch` and three output expressions for the outputs: `next state`, `Earliest Switch`, and `ISSM1stable`. A special symbol "$-$" in a table cell denotes "don't care" and matches anything. In ACL2, a table is formally defined as a constant containing a set of input

Current State:	M1	M1	M1	M2	M2	M2
ISSM1	0	0	1	1	1	0
timestamp ≥ Earliest Switch	0	1	-	0	1	-
Next State:	**M1**	**M2**	**M1**	**M2**	**M1**	**M2**
Earliest Switch	Earliest Switch	timestamp + 2000	Earliest Switch	Earliest Switch	timestamp + 2000	Earliest Switch
ISSM1 stable	**1**	**0**	**1**	**0**	**1**	**0**

(a) ISSM1 table

```
(def-table issm1-table
  :inputs (current-state      ; the name of the current state
           EarliestSwitch     ; a state variable, a time in milliseconds
           ISSM1              ; an input (boolean)
           timestamp          ; an input, in milliseconds
           )
  :case-expressions (current-state
                     (bool-to-bit ISSM1)
                     (bool-to-bit (<= EarliestSwitch timestamp)))
  :outputs (next-state EarliestSwitch ISSM1stable)
  :cases ((("M1" 0 0) ("M1" EarliestSwitch     t))
          (("M1" 0 1) ("M2" (+ timestamp 2000) nil))
          (("M1" 1 -) ("M1" EarliestSwitch     t))
          (("M2" 1 0) ("M2" EarliestSwitch     nil))
          (("M2" 1 1) ("M1" (+ timestamp 2000) t))
          (("M2" 0 -) ("M2" EarliestSwitch     nil)))
  :hyps ((member-equal current-state '("M1" "M2"))
         (booleanp ISSM1)))
```

(b) Define the ISSM1 table in ACL2

Fig. 6: Table specification for high-assurance synthesis

variables, a list of case-expressions over the input variables, a list of output variables, and a list of cases. Each case contains two parts: a list of values for the case expressions and a list of output expressions over the input variables.

We assign meaning to the tables by defining in ACL2 a generic `apply-table` function that takes two arguments: an *input map* from input variables to their values and a *table* constant. This function computes the values of the case expressions for the given table using the given input map. It examines the cases in the table, looking for a match with the computed case expressions, computes the output expressions for the matching case, and returns the outputs. A macro `def-table` is used to define and to check tables. It defines the ACL2 constant for the table and a specialized variant of `apply-table` that naively applies the given table to generate the outputs from the inputs. It also calls the ACL2 theorem prover to show that the table is complete and unambiguous (in every input scenario, exactly one case applies). Figure 6b shows the ACL2 `def-table` macro for defining and checking the table in Figure 6a. To support very wide tables, the `:cases` in the call to `def-table` are given in rows, not columns. Another slight difference from Figure 6a is the use of `t` and `nil`, indicating true and false outputs, rather 1 and 0.

The formal ACL2 specification for each component is defined via the specialized `apply-table` function.

To derive optimized ACL2 code from the formal specification for each component, we apply the APT `simplify` transformation. This unrolls and expands the specialized `apply-table` function that iterates over the cases to look for a match. The result of this partial evaluation is a large nest of if-then-else expressions, corresponding to the logic encoded in the particular table given to `apply-table`, together with a proof of equivalence. These nested conditional expressions are quite fast to execute but would be tedious and error prone to define by hand. The proofs done by the `simplify` transformation ensure that it correctly encodes the decision logic specified declaratively in the table.

The generated versions of the Monitor Selector and Contingency Manager are represented as executable ACL2 code (a subset of Common Lisp). We connect them to the rest of the system using hand-written wrapper code.

We have also generated Java code for the components by applying a Java code generator for ACL2 to the optimized and verified ACL2 code derived via APT transformations. The generated Java code is not yet verified. We are considering two approaches to verify the Java code. One is formalizing the semantics of Java and constructing a proof along with the code. The other is compiling the Java code to bytecode and then lifting the bytecode into logic and proving equivalence using Kestrel's Axe toolkit [15].

The formal synthesis approach we have taken to construct the safety components has the following advantages:

- The high-level specifications are concise and declarative. They formalize the requirements of the components.
- The generated ACL2 code is efficient and provably correct.
- With this approach we can easily adapt to changes to the requirements. First we modify the high-level specification to capture the changes to the requirements. Then we re-apply the APT transformations to the changed specification. This will automatically generate new provably correct code that satisfies the updated requirements.

IV. Results

To evaluate performance of the run-time assurance architecture, testing was performed using all six LEC variants in a variety of environmental conditions with and without the assurance architecture components. Faulty or degraded LECs were simulated by operating in conditions outside of the LEC training data set. Data was collected for the following configurations:

1) Nominal LECs without run-time assurance software. This is the baseline configuration.
2) Faulty LECs performance without run-time assurance software. These tests expose scenarios in which a faulty LEC results in unsafe system behavior.
3) Nominal LECs with run-time assurance software. These tests are intended to identify false alarms (unnecessary intervention) from the run-time assurance architecture.

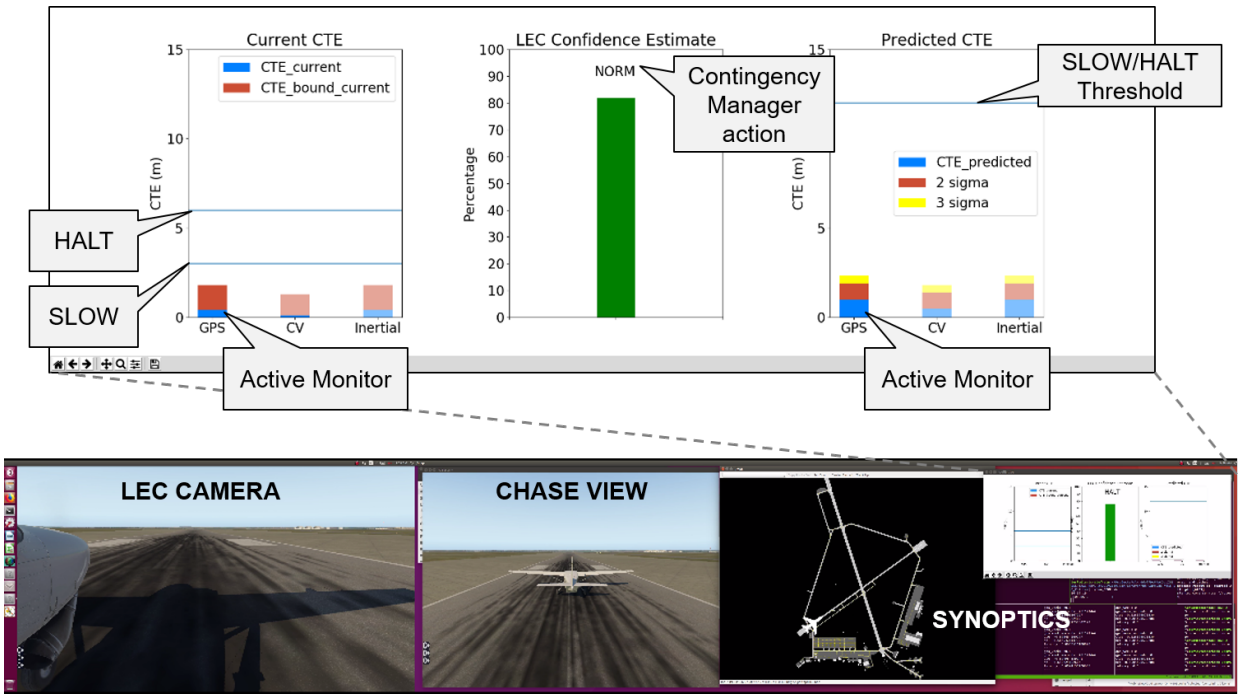

Fig. 7: Simulation Results and Monitor Synoptics Display

4) Faulty LECs with run-time assurance software. These tests are intended to determine whether the run-time architecture maintains system safety in the presence of a degraded LEC.

In all we evaluated 46 different scenarios. This allowed us to assess baseline performance, intervention of the assurance architecture in the presence of LEC errors, and absence of unnecessary intervention (false alarms). A screenshot of the demonstration video including the synoptic display of the monitor outputs is shown in Figure 7. In addition to the LEC camera input and a chase view of the aircraft, outputs from the run-time monitors and the current contingency manager action are displayed. On the left side, the current CTE estimate and error bound is displayed for each monitor. The right side shows the predicted stopping position of the aircraft given its current speed and braking performance, as computed by each monitor.

We found that the architecture performed in accordance with expectations in all scenarios. In every case where a faulty LEC caused the aircraft to deviate from the required center-line tracking performance, the assurance architecture detected the condition and slowed or halted the aircraft. At no time was the aircraft allowed to depart from the paved runway. Furthermore, we did not observe any false alarms, meaning that the architecture never intervened when the aircraft was performing within requirements and correctly tracking the runway centerline.

For example, in one scenario the LEC trained with morning-only data was tested in clear conditions at 4:00 p.m. This led to the aircraft departing from the runway after approximately two minutes. When the scenario was repeated with the run-time assurance architecture active, the aircraft was first slowed when it began to deviate from the centerline, then briefly halted, and then allowed to resume normal operation using the LEC guidance. Later, at a runway crossing, the aircraft deviated more severely and was halted to prevent it from leaving the runway.

It is important to remember that there are scenarios in which a false alarm may be expected. If the runway markings are obscured for an extended period (by water or skid marks, for example), then the CV monitor will not be able to reset the GPS monitor, and its error bound will continue to grow until it no longer has confidence in the position of the aircraft. This will trigger a slow or halt action, regardless of the actual position of the aircraft. However, under these conditions the LEC itself is unlikely to be performing very well.

AADL models for the run-time assurance architecture, the Resolute assurance case, and videos of the demonstration are available at the project website [16].

V. Conclusion

In this project we have explored run-time monitors that observe LEC inputs, outputs, and internal state, and also monitors that directly observe system safety (as shown here in the TaxiNet demo). We constructed a run-time assurance architecture with multiple monitors to enhance availability. The monitors were based on independent sensors already present on the aircraft: GPS, IRS, and video camera. We also included high-assurance safety components synthesized from formal specifications that select the best monitor and compute any required safety intervention. The system was tested with a variety of LECs under a wide range of operating conditions to demonstrate the ability of our run-time assurance architecture

to maintain safety without unnecessary intervention (false alarms).

The run-time assurance approach works best when it is possible to clearly distinguish requirements for system safety and performance, and the functions responsible for each. For example, a complex planning system may be used to compute a desired vehicle trajectory, but if safety is defined by staying within a prescribed geo-fence, this is simple to monitor using GPS. However, it is not always possible to monitor the variables or conditions needed to detect safety violations. And in some cases it is not obvious how to create a safe backup function that is less complex (and easier to verify) than the complex function to be bounded. But where the necessary conditions are satisfied, run-time assurance architectures based on ASTM F3269-17 can be a useful means for safely bounding LEC behavior.

Acknowledgments.: The authors wish to thank our colleagues James Paunicka, Matthew Moser, Alex Chen, and Dragos Margineantu for their support during integration and testing on the BR&T autonomy platform. This work was funded by DARPA contract FA8750-18-C-0099. The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] J. An and S. Cho. Variational autoencoder based anomaly detection using reconstruction probability. 2015.

[2] ASTM F3269-17. Standard practice for methods to safely bound flight behavior of unmanned aircraft systems containing complex functions, 2017.

[3] S. Bhattacharyya, D. Cofer, D. Musliner, J. Mueller, and E. Engstrom. Certification considerations for adaptive systems. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 270–279, 2015.

[4] T. Byun and S. Rayadurgam. Manifold for machine learning assurance, 2020.

[5] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. Cofer. Input prioritization for testing neural networks. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 63–70, 2019.

[6] R. Chalapathy and S. Chawla. Deep learning for anomaly detection: A survey, 2019.

[7] D. Cofer, A. Gacek, J. Backes, M. W. Whalen, L. Pike, A. Foltzer, M. Podhradsky, G. Klein, I. Kuz, J. Andronick, G. Heiser, and D. Stuart. A formal approach to constructing secure air vehicle software. *IEEE Computer Magazine*, 51, Nov. 2018.

[8] DARPA. Assured Autonomy. https://www.darpa.mil/program/assured-autonomy.

[9] C. Doersch. Tutorial on variational autoencoders, 2016.

[10] A. G. et. al. Resolute: An assurance case language for architecture models. In *HILT 2014*, pages 19–28, New York, NY, USA, 2014. ACM.

[11] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. Addison-Wesley Professional, 1st edition, 2012.

[12] A. Gacek, J. Backes, M. Whalen, L. G. Wagner, and E. Ghassabani. The jkind model checker. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 20–27. Springer, 2018.

[13] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett. The marabou framework for verification and analysis of deep neural networks. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 443–452. Springer, 2019.

[14] Kestrel Institute. APT: Automated Program Transformations. https://www.kestrel.edu/home/projects/apt/, 2020.

[15] Kestrel Institute. Axe. https://www.kestrel.edu/home/projects/axe/, 2020.

[16] Loonwerks. AAHAA: Architecture and Analysis for High-Assurance Autonomy. http://loonwerks.com/projects/aahaa.html.

[17] Matt Kaufmann and J Strother Moore. ACL2 Version 8.3. http://www.cs.utexas.edu/users/moore/acl2/, 2020.

[18] E. Nalisnick, A. Matsukawa, Y. W. Teh, and B. Lakshminarayanan. Detecting out-of-distribution inputs to deep generative models using typicality, 2019.

[19] Petovello, Mark. Inside GNSS: What is the Carrier Phase Measurement. http://www.insidegnss.com/auto/julaug10-solutions.pdf, 2010.

[20] RTCA DO-178C. Software considerations in airborne systems and equipment certification, 2011.

[21] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[22] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. Heimdahl, and S. Rayadurgam. Your "what" is my "how": Iteration and hierarchy in system design. *IEEE Software*, 30(2):54–60, 2013.