Flight Test of a Collision Avoidance Neural Network with Run-Time Assurance

Darren Cofer, Ramachandra Sattigeri, Isaac Amundson, Junaid Babar, Saqib Hasan
Collins Aerospace
{first.last}@collins.com

Eric W. Smith, Karthik Nukala
Kestrel Institute
{eric.smith, nukala}@kestrel.edu

Denis Osipychev, Matthew A. Moser, James L. Paunicka, Dragos D. Margineantu, Lucca Timmerman, Jordan Q. Stringfield
Boeing
{first.mi.last}@boeing.com

***Abstract*—Our team is developing assurance technologies that can support the use of machine learning in the design of safety-critical aircraft systems. These capabilities have been integrated on Boeing's Autonomy Testbed Aircraft to show that they can provide evidence of correct operation and safety guarantees needed by real aircraft. We have applied run-time assurance along with formal methods synthesis, modeling, and analysis tools to an airborne collision avoidance system based on a neural network. This system was demonstrated in flight and shown to correctly monitor neural network operation and intervene when needed to prevent violation of the "remain well clear" safety requirement relative to an intruder aircraft.**

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

I. Introduction

Aircraft systems have requirements for airworthiness certification that present barriers to the use of machine learning technologies such as neural networks. Showing that a component or system is correct and does no harm through behaviors that were unintended by designers or unexpected by operators is a critical aspect of the certification process. In a typical machine learning application, much of the complexity and design information resides in its training data rather than in the actual models or software produced. This means that it is generally not possible to determine the correctness of a *learning enabled component (LEC)*, such as a neural network, by examining its implementation or tracing specific design elements back to requirements.

Our team is developing assurance technologies that can support the use of machine learning in the design of safety-critical aircraft systems. These capabilities have been integrated on Boeing's autonomy demonstrator aircraft to show that they can provide evidence of correct operation and safety guarantees needed by real aircraft. In previous work [5] we have used a run-time assurance (RTA) architecture to ensure the safety of an autonomous neural network-based aircraft taxiing application. In our current work we have applied RTA along with formal methods modeling, synthesis, and analysis tools to an airborne collision avoidance system based on a neural network. This system was demonstrated in flight and shown to correctly monitor neural network operation and intervene when needed to prevent violation of the "remain well clear" safety requirement relative to an intruder aircraft. Thirty-eight test conditions were flown with various collision course geometries to test the robustness of the neural net trajectory generator and the RTA mitigations.

The RTA architecture includes a run-time monitor that provides an independent assessment of the avoidance flight plans generated by the neural network and a safe (but less optimal) backup planner. The results of the assessment are evaluated by a decision logic component which selects (based on a tabular specification of safety rules) a flight plan that will ensure safe flight. The core decision logic code is synthesized from a formal specification, with most of the synthesis steps producing machine-checked proofs of their correctness. The RTA architecture has been modeled in the Architecture Analysis and Design Language (AADL) and formally analyzed to show that it maintains the system safety requirements.

Remain-well-clear and collision avoidance capabilities are critical to safe flight of autonomous military and commercial aircraft. These capabilities can also be valuable for enhancing the safety of piloted aircraft by providing pilot cueing. Run-time assurance approaches, with run-time monitoring of machine learning software functions integrated with contingency management functionality, will enable safe use of neural networks and enable new autonomous capabilities for aircraft.

In this paper we will discuss:

- The assurance challenges associated with the use of LECs in safety-critical aircraft applications
- The autonomy framework and aircraft used to demonstrate the collision avoidance neural network capability
- The run-time assurance architecture developed to guarantee that any potential unintended behaviors in the neural network do not impact safety
- The formal methods assurance technologies applied within the architecture, including analysis of the architecture design and synthesis of decision logic from a formal specification
- Flight testing, results obtained for various test scenarios, and lessons learned from the demonstration

II. Assurance Challenges

For software in commercial aircraft, DO-178C [16] provides the latest version of guidance regarding software aspects of certification and is used by the aviation industry and regulators as a means of compliance with airworthiness regulations.

At a high level, DO-178C helps manufacturers achieve two main goals: 1) demonstrate that software complies with its requirements (intended functionality), and 2) show that it does nothing unexpected (unintended functionality). Unintended functionality or unintended behavior can therefore be defined as software behavior than cannot be traced back to any requirement.

LECs and their software implementations break many of the assumptions that are the basis for current certification processes. An LEC design is created through training based on large amounts of example data. This means, for example, that individual elements in a neural network (weights, connections, or activation functions configured during training) do not represent design choices that can be traced back to specific requirements. Therefore, the design intent cannot be inferred from an examination of an LEC model or its software implementation.

DO-178C fundamentally relies on requirements-based testing (or verification) and structural coverage metrics, and works extremely well to show that a traditional software development process correctly implements a set of requirements. Structural coverage metrics were constructed with the understanding that much of the complexity of traditional software is manifested in the logical decisions that are being implemented. This logic should be traceable to specific software requirements. When requirements-based tests fail to exercise part of the software logic as revealed by structural coverage metrics, it is reasonable to conclude that something is amiss (either a missing requirement or some unintended behavior). Since neural networks do not primarily implement logical decisions, structural coverage can usually be achieved with one test case (or possibly a small number of them) [13]. Therefore, current structural coverage metrics are not helpful in identifying and eliminating unintended behaviors. Several alternative coverage metrics have been proposed for neural networks, but so far none has been shown to provide equivalent assurance.

Since it is difficult to demonstrate assurance by examining the LEC design (as is assumed by existing certification processes), other approaches based on run-time monitoring and enforcement may be effective.

III. Autonomy Framework

Our experiments are carried out on Boeing's Autonomy Testbed aircraft and simulation environment. This testbed is used for the development of the collision avoidance LEC, the RTA architecture for ensuring safety, and for flight testing and demonstration.

A. Autonomy Testbed Aircraft

The Boeing Autonomy Testbed Aircraft is a Cessna Caravan 208B, tail number N208BX (Figure 1). This platform is currently serving as a test bed for the DARPA Assured Autonomy program air domain experiments. The Testbed is optionally piloted and serves as a means to demonstrate commercially viable technologies leading to autonomy. It is a research and development vehicle able to operate in commercial airspace that is built on open-source middleware with in-house developed guidance and control technologies leveraged from across the Boeing enterprise. The Testbed includes a full "Iron Bird" fixture for hardware-in-the-loop evaluation in which new autonomy technology can be fully integrated and tested before flight. With this Testbed Boeing has demonstrated autonomy firsts including in-air detect and avoid, ADS-B transponder-based route planning for strategic avoidance, and fully autonomous ground taxi.

Fig. 1: Boeing Autonomy Testbed Aircraft, Cessna Caravan 208B

Detect and Avoid (DAA) mission operating and performance standards are defined in RTCA document DO-365 [1]. The standard provides guidance for interactions of Unmanned Aircraft Systems (UAS) in the National Airspace System, requirements for safe operation of aircraft during encounters including separation distance minimums for remaining well clear of aircraft and avoiding mid-air collisions, and proper aircraft equipage to achieve safe detect and avoid operations. The assurance challenge posed in our work focuses on the Autonomy Testbed Aircraft flying in the vicinity of another "intruder" airplane, where the test flight software includes a Boeing-developed LEC to generate an avoidance flight plan for the Testbed to remain well clear of the intruder aircraft as defined in DO-365. The underlying assurance technology montors the Boeing LEC in order to assess the avoidance trajectory from the LEC and guarantee safety.

Figure 2 shows a block diagram of the autonomy system framework as deployed on the Testbed aircraft. System elements include:

- ADS-B – The primary sensor for perceiving the airspace is the Automated Dependent Surveillance Broadcast (ADS-B) system providing detection information on nearby aircraft (intruders) to the other system functions.
- Avoidance Alerts – Evaluates potential future traffic conflicts and issues "alerts" to the Avoidance functions as specified in DO-365.

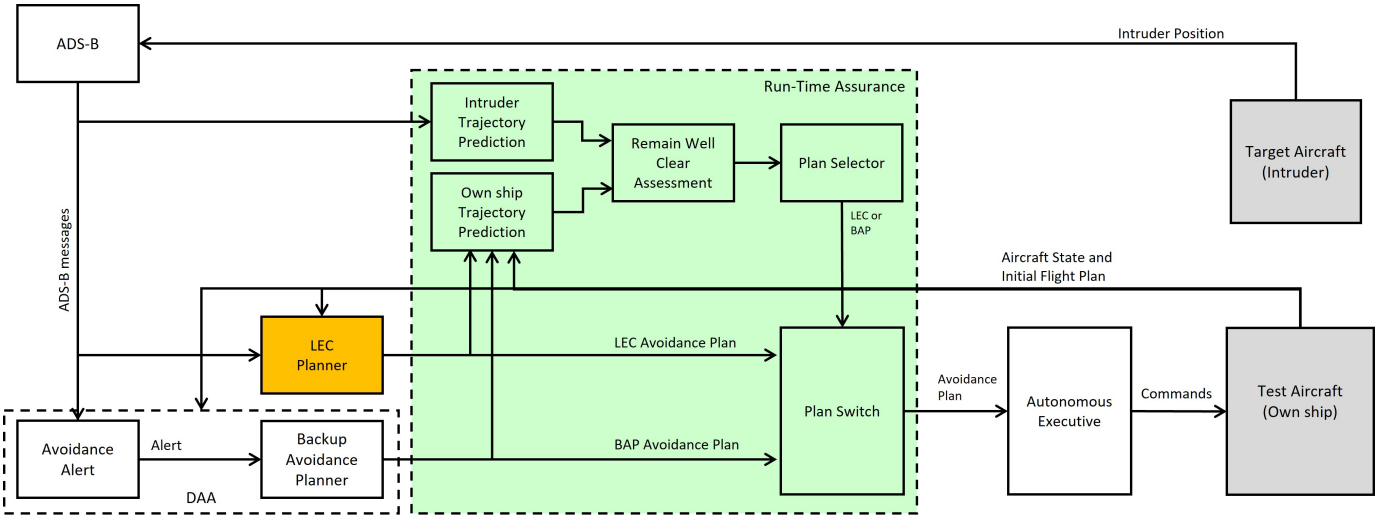

Fig. 2: Collision avoidance system on the autonomy testbed aircraft with run-time assurance components in green

- LEC Planner – Neural network system trained offline through reinforcement learning to generate avoidance flight plans satisfying DAA requirements. Training and operation is described in the following subsection.
- Backup Avoidance Planner – A trusted but less optimal backup planner that provides waypoint navigation paths to avoid airspace incursion. Avoidance computation method is virtual predictive radar which is designed to provide maximum "safe passage timed corridors." Avoidance path terminates back on the original flight plan.
- Run-Time Assurance – Run-time monitoring, prediction, and assessment systems that guarantee the selection of a safe flight plan for the aircraft. Operation and assurance are described in Sections IV and V.
- Autonomous Executive – Constructs and manages execution of the vehicle flight plan and contains a function to "splice in" avoidance guidance waypoint plans into the original flight plan.

B. LEC Training with Reinforcement Learning

The LEC that generates the collsion avoidance flight plan is trained offline with a Reinforcement Learning (RL) policy model. RL is a sequential policy optimization method that solves the task using the "learning by doing" concept and requires continuous data-rich interaction with the environment [18]. Our avoidance policy is trained on a surrogate task to provide the large number of interactions needed to achieve good performance. This surrogate environment developed by Boeing is a lightweight Python environment integrated with the OpenAI GYM framework [4].

The learned policy minimizes the risk of collision by providing continuous control commands in the surrogate environment. These commands are converted into a geometric trajectory (a sequence of waypoints) using the surrogate environment and a Robot Operating System (ROS) interface (Figure 3).

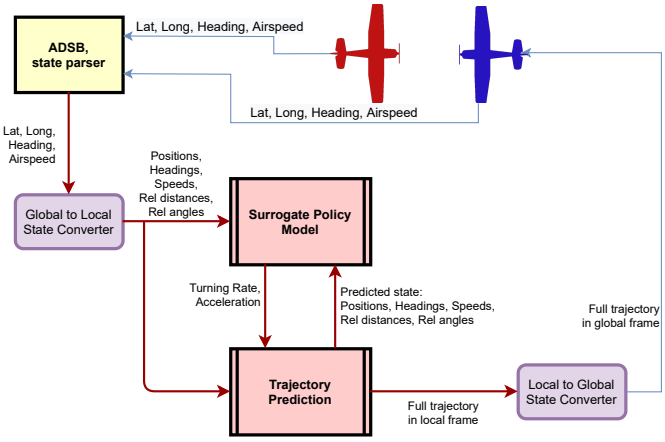

Fig. 3: System diagram of the RL environment

Our surrogate environment is a simplified 2D obstacle avoidance problem that mimics the real task (air traffic conflict resolution). The environment simulates the movements of two aircraft on a 20x20 km square. The controlled agent (representing the Autonomy Testbed Aircraft) has to go around the intruder, provide minimal horizontal separation, and merge back to the next safe waypoint from the original route before the simulation ends.

The surrogate dynamics imitate the dynamics of the Autonomy Testbed Aircraft. Both aircraft are represented by a simple dynamic model assuming massless kinematics and using Dubin's vehicle model for turn dynamics. All the observation and control values are normalized to [-1..1] for the RL agent. To make sure the agent generalizes the problem, we rewrap the observations and focus only on relative positions rather than absolute. Observations consumed by the agent are heading, airspeed, distance to goal, tracking angle to goal, intruder heading, intruder airspeed, distance to intruder, and tracking angle to intruder.

C. RL Policy Agent

The RL policy agent learns the task by interacting with the simulation and iteratively updating the parameters of the policy model using Stochastic Gradient Descent (SGD) optimization. We approximate the policy with a multi-layered perceptron shown in Figure 4.

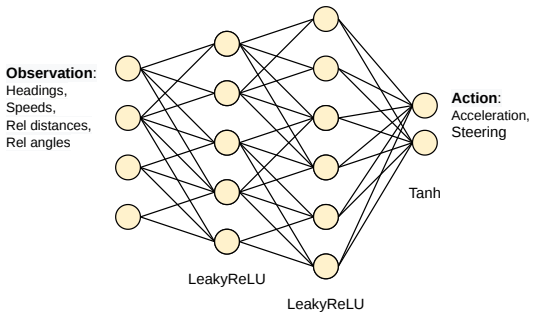

Fig. 4: Neural network function approximation used for the policy model consists of 2 hidden layers, 256 neurons each

To solve the optimization problem as Markov Decision Process (MDP), we refactor it into Markovian states s, transitions $T(s'|s,a)$, and transition reward $R(s'|s,a)$. The state of the system (including both agent and intruder) is fully observable, assumes the perfect knowledge, and is enough to describe the Markovian state of the MDP system. The state of the agent is described as

$$s = \{v_a, \psi_a, v_i, \psi_i, \beta_i, d_i, \beta_g, d_g\}$$

where v_a is agent speed, ψ_a is agent heading, v_i is intruder speed, ψ_i is intruder heading, β_i is angle to intruder, d_i is distance to intruder, β_g is angle to goal, and d_g is distance to goal.

The optimization is set to find the optimal policy $\pi^*(s)$ as a set of state-action mappings that maximizes the expected reward $V(s)$ [18].

$$\pi(s) = P(a|s) \tag{1}$$
$$\pi^*(s) = \arg\max_{\pi} V^{\pi}(s) \tag{2}$$
$$= \arg\max_{a} \left(R(s,a) + \gamma T(s'|s,a)V(s')\right) \tag{3}$$

The value of the state is the expected future reward accumulated over the trajectory and defined by the Bellman function as:

$$V(s) = \mathbb{E}[R|s,\pi] \tag{4}$$
$$= \sum_{s'} T(s'|s,a)\left(R(s'|s,a) + \gamma(V(s'))\right) \tag{5}$$
$$= R(s'|s,a) + \gamma \sum_{s'} T(s'|s,a)V^{\pi}(s') \tag{6}$$

$$V^*(s) = \max_{a} \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a)V^*(s')\right) \tag{7}$$

The RL policy model is based on the Actor-Critic architecture that helps to improve the stability of the training [18]. The SGD-based update for Actor θ and Critic w network is:

$$\delta = R_{t+1} + \gamma\hat{V}(s_{t+1}, w) - \hat{V}(s_t, w) \tag{8}$$
$$w \leftarrow w + \alpha\delta\nabla\hat{V}(s,w) \tag{9}$$
$$\theta \leftarrow \theta + \alpha\delta\nabla \ln \pi(a|s,\theta) \tag{10}$$

The core functionality of the RL agent incorporates the Stable Baselines library, a very reputable fork of OpenAI Baselines [7]. For the exploration policy and update steps, this work used the Proximal Policy Optimization (PPO) algorithm that becomes the state of the art in continuous-action agents [17].

D. LEC Integration on Testbed Aircraft

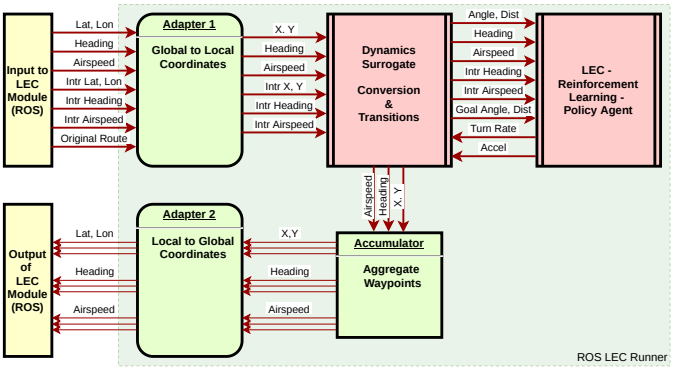

Fig. 5: System diagram of the ROS-LEC node showing the integration of the learning-enabled component (LEC) to the aircraft using the ROS interface

System integration is done using the Robot Operating System (ROS) interface [14]. This allows unifying the interfaces to the high-fidelity simulation and to the physical aircraft demonstrator. The ROS-LEC agent, shown in Figure 5, aggregates data from different domains and provides important utilities to the system. Its job is designed as follows:

- receive and accumulate ROS messages regarding the own-ship state, intruder's state, traffic alerts, GPS-SRS transformation data,
- translate ADS-B and GPS positioning data to local coordinate frame,
- extract the goal location from the original route,
- re-wrap the observations into the agent-specific input format,
- iteratively run the agent to get the corrective actions,
- iteratively run the surrogate environment to receive the transitions,
- form a corrective trajectory and check if the trajectory is good,
- translate the trajectory from local coordinate frame to global lat-long waypoints,
- publish the trajectory as a ROS message.

On an external request, the ROS-LEC agent generates a single avoidance flight plan and publishes it as a ROS message.

The number and spacing of waypoints are configurable. For our experiment, the flight plan consists of 20 waypoints spaced 20 seconds apart. The last waypoint is taken from the original route, and 19 waypoints are generated by the policy. This allows linking the waypoints by a unique index and preserves the indices of the original route. Because of the large time step between the waypoints, the policy and the surrogate have to be evaluated 20 times to make a single waypoint. The total response time of the system is less than 60 msec for the complete trajectory.

IV. Run-Time Assurance

Run-time assurance architectures add high-assurance components to the system to ensure that a complex or difficult-to-verify component (such as our LEC) cannot cause unsafe or unintended system behaviors. Run-time monitors continuously check input or output values of the LEC to assess safety and can intervene to switch to a backup function that is proven to be safe.

In this project Collins Aerospace engineers developed a run-time assurance architecture based on the ASTM F3269-17 standard for bounded behavior of complex systems [3], also known as a simplex architecture [15]. The standard provides guidance for mitigating unintended behaviors through the use of run-time monitors. The LEC may still produce unintended behavior, but the architecture ensures that there will be no impact on system safety. This approach uses the verified properties of the architecture, run-time monitor, and safety backup function to justify a reduced level of criticality for the LEC.

The run-time assurance architecture is illustrated in the green box in Figure 2. Incoming ADS-B messages are assessed by the DAA subsystem. When an avoidance alert is generated, the Backup Avoidance Planner and the LEC Planner both generate avoidance flight plans. The RTA monitors perform a "remain well clear" (RWC) assessment on both plans, determining whether either plan will result in a violation of the DO-365 separation requirements. Based on the assessments, Plan Selector then chooses one of the two plans and the Plan Switch publishes the chosen flight plan.

A. Trajectory Prediction

Trajectory prediction over a defined prediction horizon (in time units) consists of prediction of both the intruder and own-ship trajectories based on underlying assumptions of the intruder's future velocity and the own-ship's ability to track the avoidance flight plans from the LEC and the Safe Backup Planner. For the application in consideration, the prediction horizon was set to 180 seconds under the assumption that the avoidance flight plans produced would be frozen for this time period. In cases where the avoidance plans can be generated more frequently, the prediction horizon can appropriately be reduced which should improve the accuracy of intruder trajectory prediction and allow for more confidence in the RWC evaluation.

For the intruder, each incoming ADS-B message is processed as a measurement to a tracking filter designed as per Appendix D in [2]. The tracking filter assumes that the intruder aircraft is either in a constant velocity (CV) mode or in a constant speed turn (CT) mode. The multiple-model tracking filter is designed to be robust to basic errors in the ADS-B message data (such as data with too small or large uncertainties), missing data or repeat data, but mostly the ADS-B data is considered a trustworthy source of information of the intruder 3D position. The tracking filter produces estimates of the intruder current position and velocity along with their respective uncertainties, and with a prediction of the specific mode of the intruder behavior (CV/CT). The trajectory prediction propagates the estimate from the tracking filter forward in time by assuming the aircraft continues in its current mode (CV/CT) with a growth model in the longitudinal velocity uncertainty and a cross-track position uncertainty, with the cross-track position uncertainty capped to a maximum value. The uncertainties in the intruder trajectories can be visualized as ellipses whose semi-major/semi-minor axes are aligned with the longitudinal/normal to velocity vector and whose lengths are proportional to the uncertainties along/normal to the velocity vector. These assumptions allow to account for decaying confidence over longer prediction horizons.

The own-ship trajectory predictions along the avoidance flight plans assume a kinematic model of the own-ship behavior using performance parameters such as maximum bank angle, roll-rate, longitudinal acceleration, etc. The trajectory prediction assumes that the first waypoint of the avoidance flight plans lies directly in the path of the current velocity vector. The own-ship trajectory consists of constant ground speed segments between waypoints and constant speed arcs connecting line segments between consecutive waypoints. As the actual aircraft is flying at a constant airspeed, and not ground speed, deviations between the predicted and actual aircraft trajectories are expected. These deviations are accounted for by a fixed tracking error bound that is configurable. The tracking error bound and vehicle size parameter are added to the specified RWC separation distance to produce a conflict radius threshold around the own-ship.

The trajectory prediction models are run internally at a high-rate (over 10 Hz) for both the intruder and own-ship, but the output trajectories are sampled at 1 Hz to produce time-stamped discretized trajectory points which are evaluated for conflicts as discussed in the next sub-section.

B. Remain Well Clear Assessment

The RWC assessment consists of evaluating time-stamped samples of the predicted trajectory of the intruder and the own-aircraft trajectory along each produced avoidance flight plan. The evaluation considers the intersection of the uncertainty ellipse around an intruder predicted trajectory sample with the conflict circle around the corresponding own-ship predicted trajectory sample. The probability of intersection is determined using the analytical approximations documented in Section IV

of [8]. If the probability exceeds a configurable threshold, then the flight plan is determined to be unsafe.

In addition to the determination of safety, the closest point of approach (CPA) between the own-ship and intruder predicted trajectories is calculated along with the time to CPA. The CPA metrics allow the Plan Selector decision logic component to choose between plans that are both marked unsafe by the RWC assessment. These metrics allow for a comparison with the actual measured CPA metrics during simulation and test flights as a way to evaluate the run-time monitor performance.

C. Plan Selector

A critical component of the RTA architecture is the Plan Selector, which selects the flight plan to be used based on formally verified decision logic. This decision logic specifies the rules that determine whether the LEC flight plan or the Backup Avoidance flight plan (BAF) should be selected based on the results received from the RWC Assessment component. The decision logic is specified in a tabular format, shown in Figure 6. After running the logic, the Plan Selector sends its decision to the aircraft Plan Switch.

When a flight plan assessment is received, it is evaluated based on five variables, including the RWC metrics: plan validity (whether a plan has been received), plan safety, whether the time of closest point of approach (tCPA) is greater than 179 seconds (the end of the planning horizon), comparison of the predicted miss distance (pmd) between the intruder and own-ship, and whether three seconds has elapsed since the receipt of the first plan.

If only one plan has been received by the three second timeout, we select (publish) that plan by default. If we receive two plans and only one is safe, we select the safe plan. If neither plan is safe, we select the one with the larger predicted miss distance. If both plans are safe, the LEC plan is usually selected. However, there is some additional tie-breaking logic. If tCPA is beyond the planning horizon for one of the plan, this means that we don't really know what its true CPA is so we select the other plan. If tCPA is beyond the planning horizon for both plans, we fall back to selecting the plan with the greater predicted miss distance.

Different selection logic could be specified depending on vehicle and program requirements, and indeed our specification evolved over the course of the project. However, the formal synthesis approach (described in the next section) ensures that we can quickly regenerate a correct implementation of the specification.

V. Assurance Technologies

In addition to the RTA architecture itself, we have applied formal methods technologies to several aspects of the design to help ensure system safety. This includes analysis of the AADL system architecture model, synthesis of the plan selector decision logic from a formal specification, and production of an assurance argument documenting the evidence and rationale for safety of the RTA approach.

A. Architecture Verification

One of the steps for design-time assurance of the RTA architecture is verifying the architecture satisfies its high-level requirements. Traditionally, requirements verification has been achieved using a combination of directed testing methods and manual review. However, model-based specification enables a more rigorous approach to verification via formal methods analysis. With both the requirements and architecture represented in formal (well-defined, unambiguous) notations, satisfiability modulo theories (SMT) solvers can be employed to determine whether there is any possible sequence of inputs that will violate a requirement. Furthermore, failure by the solver to find a counterexample is essentially equivalent to a mathematical proof that the requirement can never be violated.

In architectural models, we can represent high-level requirements as assume-guarantee contracts on components. *Guarantees* are statements about a component's outputs which will always hold as long as stated *assumptions* are valid. When designing an architecture that includes multiple components, it is imperative to verify that a system's subcomponent contracts satisfy the overall system contract, as well as whether a component's assumptions are valid with respect to the specified upstream guarantees and the environment. We use the Assume Guarantee Reasoning Environment (AGREE) [19] to specify and analyze component contracts in our run-time assurance architecture. AGREE is a plugin for the Open Source AADL Tool Environment (OSATE), enabling contracts to be specified directly on AADL model components and analyzed within the modeling environment.

The main objective of our AGREE analysis of the collision avoidance system was to verify that the RTA architecture is guaranteed to publish only safe flight plans. As illustrated in Figure 7, we annotated the AADL model with assume-guarantee contracts for each component in the architecture, and AGREE was able to verify the high-level property under the assumption that the backup avoidance flight plan is always safe. There are four guarantees shown in Figure 7:

- An ADS-B intruder conflict results in a BAF plan being generated.
- An ADS-B intruder conflict results in an LEC plan being generated.
- An ADS-B intruder conflict results in a safe plan being selected by RTA under the assumption that BAF is always safe.
- An ADS-B intruder conflict results in a safe plan being forwarded by the Plan Switch under the assumption that BAF is always safe.

The figure also demonstrates the use of AGREE to detect *vacuous* guarantees. This refers to guarantees that are true only because the context in which the guarantee should hold never occurs. Figure 7 shows two such statements:

- RTA always selects BAF if it is safe.
- The Plan Switch always forwards BAF if it is safe.

In other words, we are checking to see if the RTA architecture ever selects the LEC plan. AGREE analysis shows that these

inhibiting	LEC valid	BAF valid	LEC safe	BAF safe	LEC tCPA > 179	BAF tCPA > 179	LEC pmd < BAF pmd	currtime > LEC.time + 3	currtime > BAF.time + 3	output
1	-	-	-	-	-	-	-	-	-	NO_PUBLISH
0	0	0	-	-	-	-	-	-	-	NO_PUBLISH
0	1	0	-	-	-	-	-	0	-	NO_PUBLISH
0	1	0	-	-	-	-	-	1	-	PUBLISH_LEC
0	0	1	-	-	-	-	-	-	0	NO_PUBLISH
0	0	1	-	-	-	-	-	-	1	PUBLISH_BAF
0	1	1	0	0	-	-	1	-	-	PUBLISH_BAF
0	1	1	0	0	-	-	0	-	-	PUBLISH_LEC
0	1	1	1	0	-	-	-	-	-	PUBLISH_LEC
0	1	1	0	1	-	-	-	-	-	PUBLISH_BAF
0	1	1	1	1	0	0	-	-	-	PUBLISH_LEC
0	1	1	1	1	1	0	-	-	-	PUBLISH_BAF
0	1	1	1	1	0	1	-	-	-	PUBLISH_LEC
0	1	1	1	1	1	1	1	-	-	PUBLISH_BAF
0	1	1	1	1	1	1	0	-	-	PUBLISH_LEC

Only one valid plan received : select that plan

Time out

Neither is safe : select the least unsafe

Only one plan is safe : select that plan

Both plans are safe : select LEC

UNLESS LEC CPA is beyond assessment horizon

OR both CPAs are beyond horizon : select the least unsafe

Fig. 6: Plan Selection logic specification (hyphens match either 0 or 1)

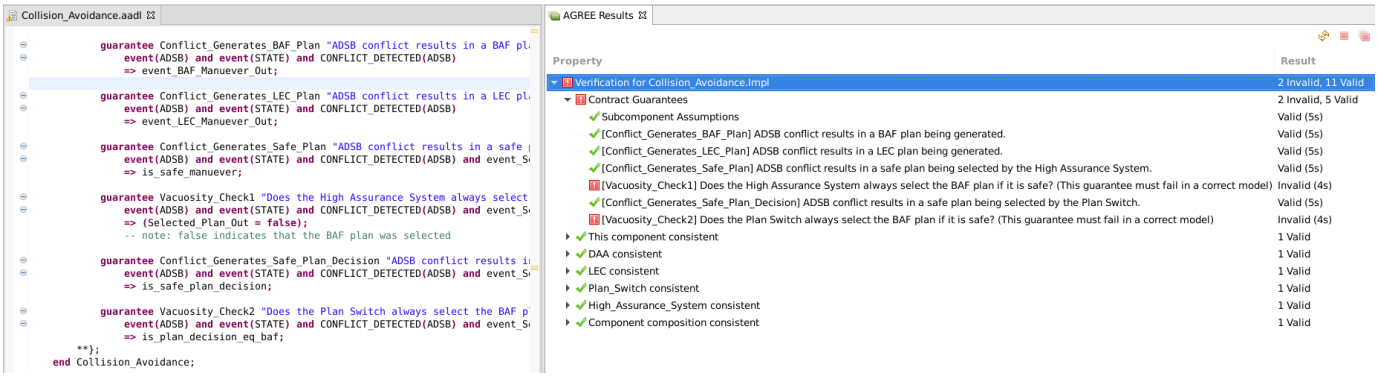

Fig. 7: Specification and checking of Run-Time Assurance properties with AGREE

statements are false, giving legitimacy to the guarantees of the RTA System and Plan Switch.

B. Decision Logic Synthesis

We formalized this decision table and applied our synthesis tools, based on the ACL2 theorem prover [12], to synthesize high-assurance decision code implementing the table logic. This process used the `def-table` machinery described in [5]. In particular, proofs are automatically performed on the table to ensure completeness (some case always applies, and an output is always produced) and unambiguity (no more than one case applies, so a unique output is always produced).

Our software synthesis process starts with an executable specification that applies our generic table evaluation procedure `apply-table` to the specific table described above. In this initial specification, the table is represented as a piece of data that is interpreted by `apply-table`. Synthesis steps will specialize the table evaluation process, building in the specific table to create a cascade of conditional expressions.

We synthesize an executable Java program in two key steps:

1) Apply APT transformations to specialize and simplify the decision logic, creating an optimized, executable function: We use Kestrel's APT (Automated Program Transformations) [9] toolkit, built on ACL2, especially the `simplify` transformation, which applies simpli-

fying rewrite rules from our library. Each of the APT transformation steps produces a proof showing the equivalence of its input and output.

2) Generate Java code using ATJ [11]: We use Kestrel's ATJ (ACL2-to-Java) Java code generator, built on ACL2, to generate Java code for integration with the Collins RTA. Extending previous work (described in [5]), ATJ now generates idiomatic and more performant Java code.

For integration into the larger system, we created a hand-written wrapper for the generated Java decision code. The wrapper receives and processes incoming plan messages from the Remain Well Clear Assessment, applies the generated decision code to the boolean plan variables, and publishes the decision to the Plan Switch. Communication is done via ROS.

Our methodology for synthesizing the decision logic using program transformations allows for quick re-synthesis when the design changes. One simply re-runs the synthesis tools using the new table-based specification as input. This flexibility was demonstrated when simulation results necessitated additional conditions and constraints on plan selection. Specifically, the table was changed to consider whether each plan's tCPA exceeds the 179 second planning horizon. This simply amounted to changing the `def-table` encoding in ACL2, with all the correctness proofs, synthesis/optimization steps, and Java code generation following immediately.

Future improvements to this methodology may include:

- Outfitting ATJ with proofs: As remarked in [5], this might involve either formalizing the semantics of Java or using Kestrel's Axe toolkit to lift Java bytecode into logic.
- Code generation for other languages (C, Python, Rust): Recent work at Kestrel was able to achieve the same synthesis presented above using the ATC C code generator [10] in place of ATJ, with the additional benefit of having proofs-of-correctness for the code generation.
- Formal synthesis of RTA-logic interface: At the time of publication, the integration of the plan selection logic with the Collins RTA requires a handwritten wrapper, which could potentially also be synthesized with proofs.

C. Assurance Argument

The assurance argument for the run-time assurance architecture is made explicit using Resolute [6]. Resolute is a tool for specifying and instantiating assurance patterns, and analyzing the resulting assurance argument to determine whether all claims are supported by evidence. Resolute is an OSATE plugin, which provides tight coupling between an assurance argument and the system architecture under evaluation. Consequently, Resolute is able to instantiate the assurance pattern with context specified within the system model, extract evidence for supporting assurance claims, and produce a "passing" or "failing" assurance argument based on whether all claims are satisfied.

Figure 8 depicts the assurance argument for the run-time assurance architecture. The full argument is not shown due to space constraints, but the general structure consists of a top-level claim supported by three sub-claims. Overall, we wish to argue that the RTA architecture ensures that a safe flight plan is published (claim G1 in the figure). In the event that the flight plan produced by the LEC is not deemed safe, we fall back on the backup flight plan produced by the BAF. We therefore include the assumption (A1) that the BAF flight plan is indeed safe.

We argue the top-level goal is achieved by providing evidence that the architecture is correct (G4), the implementation logic is correct (G3), and that an unsafe flight plan can be detected when it is produced by the LEC (G2). The architecture correctness argument comprises multiple sub-claims that focus on correct model composition with respect to requirements. As shown in the figure, one method for demonstrating correctness is AGREE verification (G14). This particular branch of the assurance argument also demonstrates Resolute's ability to evaluate evidence produced by external tools as part of evaluating the assurance argument. The implementation logic correctness argument is supported by a proof of correctness from the APT tool (G9). The safe backup sub-claim argues that the decision from the run-time monitor is safe. The evidence for this claim ensures that both the run-time monitor requirements and implementation is correct.

VI. Flight Test Results

Our flight testing plan evaluated the performance of the LEC collision avoidance planner and the RTA architecture using the Boeing Autonomy Testbed Aircraft and another (unmodified) Cessna Caravan aircraft flying as an intruder. Both aircraft executed a variety of straight-and-level trajectories headed towards a defined collision point (but with 400 feet of vertical separation for safety) in our test area over Central Washington State. The flights occurred in an airspace volume closely coordinated with Air Traffic Control at Grant County International Airport in Moses Lake, Washington. The flight test plan, and the design of all aircraft systems, underwent thorough safety reviews by the US Air Force and the Boeing Company before flight testing.

The two aircraft used in the flight testing were equipped with ADS-B In and ADS-B Out functionality. The Boeing autonomy testbed aircraft used its onboard ADS-B In system to sense the location of the intruder aircraft which was on a collision course towards the Control Point (CP), as shown in Figure 9. The DAA alerting functionality on the testbed aircraft detected the intruder at a safe distance and triggered the generation of avoidance flight plans that the testbed would fly to remain well clear of the intruder aircraft.

As described above, two avoidance flight plans were automatically generated on the testbed aircraft, one by the LEC and the other by the backup avoidance planner (which did not use a neural network). The RTA architecture assessed the two plans and determined which would be flown by the testbed to remain well clear of the intruder.

Flight testing consisted of the live execution of multiple two-aircraft test conditions. Each test condition specified the following:

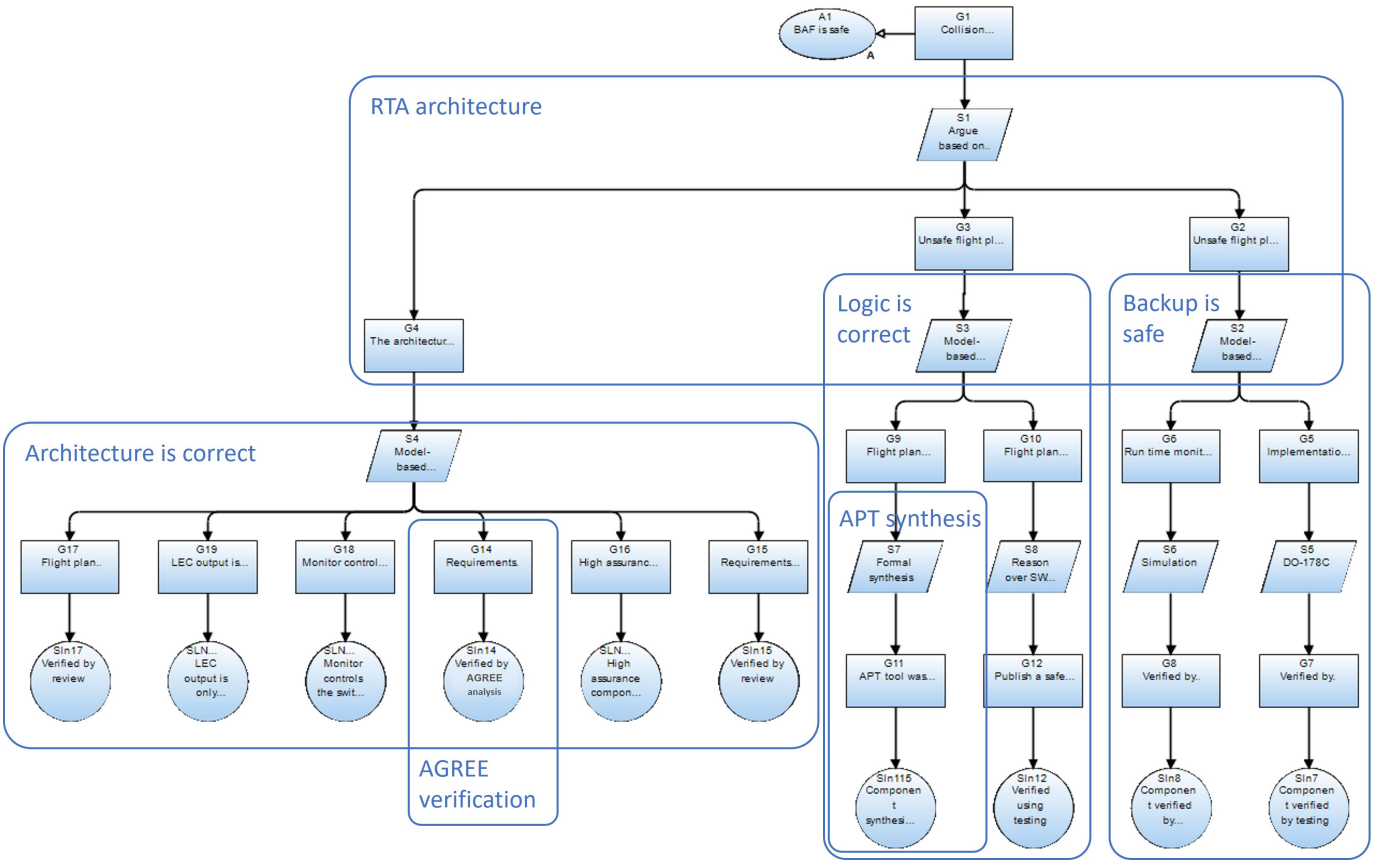

Fig. 8: Assurance Argument for run-time assurance generated by Resolute

1) A relative heading angle between the testbed aircraft and the intruder
2) The specification to use one of two LECs that were available on the testbed aircraft flight software

Multiple relative heading angles were flown, including a head-on encounter (relative heading = 180 degrees), and other relative headings in 45 degree intervals. The available LECs were termed the "good LEC" and the "bad LEC." The "good LEC" was expected to generate safe remain-well-clear trajectories, while the "bad LEC" was designed to generate unsafe trajectories, simulating an LEC producing unintended (and unsafe) behaviors.

During the numerous test conditions flown we made the following general observations:

- In test conditions where the good LEC was used, the generated avoidance flight plans resulted in safe and standards conformant remain-well-clear avoidance of the intruder. The RTA functionality successfully assessed the LEC plans as safe, which resulted in the testbed aircraft flying the LEC route.
- In test conditions where the bad LEC was used, the generated avoidance flight plans resulted in violation of the remain-well-clear avoidance criteria relative to the intruder. The RTA functionality successfully assessed the LEC plans as unsafe, which resulted in the testbed aircraft flying the route from the backup avoidance planner.

There were two interesting test conditions in which we observed unexpected results. Recall from Section IV that when both plans are assessed as safe but predicted CPA for either occurs at the limit of the prediction horizon, we prefer the plan whose CPA is actually within the prediction horizon. This situation occurred spontaneously in one of our test conditions, resulting in the RTA functionality choosing (correctly) the backup plan over the LEC.

In another test condition we were surprised to observe the RTA functionality choosing to fly the plan generated by the bad LEC. Upon further analysis, we found that both plans were assessed as safe (though the bad LEC plan was just barely safe) and in this case the plan selector logic correctly chose the LEC plan. However, during execution of the test scenario the RWC separation criteria was violated. We discovered that several factors combined to place the intruder aircaft ahead of its predicted position, resulting in separation slightly below the RWC requirement. This condition was possible in our experiment because of an initial design decision to have each of the planners produce only a single avoidance flight plan at the start of a collision encounter, with no updates for any changes that might occur during test execution.

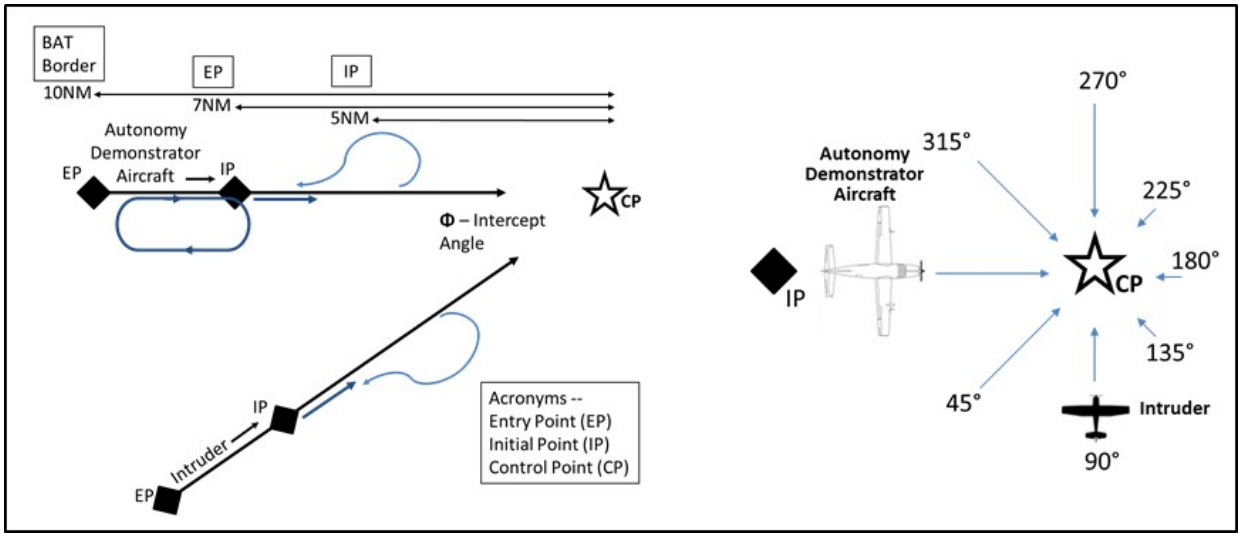

Fig. 9: Flight test plan for evaluation of LEC collision avoidance and RTA safety guarantees

VII. Conclusion

Our team has flight tested machine learning technology for aircraft collision avoidance with a run-time assurance architecture designed to guarantee safety in the presence of potential unintended behaviors. These capabilities were integrated on Boeing's Autonomy Testbed Aircraft to show that they can provide correct operation and safety guarantees needed by real aircraft. Flight testing demonstrated the ability of the RTA system to ensure that "remain well clear" separation criteria were maintained during a variety of collision encounter geometries. Formal methods and an assurance argument were used to provide evidence of the correctness of the RTA design.

Future work will extend the LEC planner and the RTA architecture to handle multiple intruder aircraft and other obstacles such as weather. We will also update the RTA architecture to actively update RWC assessment during a collision encounter and dynamically request new avoidance plans if safety has been compromised due to change in intruder aircraft behavior or the arrival of additional intruders.

Acknowledgment: The authors wish to thank our colleagues on the Boeing flight testing team and the staff at the Grant County International Airport for their support during flight testing. 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] RTCA/DO-365 Minimum Operational Performance Standards (MOPS) for Detect and Avoid (DAA) Systems, May 2017.

[2] RTCA/DO-366A Minimum Operational Performance Standards (MOPS) for Air-to-Air Radar for Traffic Surveillance, September 2020.

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

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[5] D. D. Cofer, I. Amundson, R. Sattigeri, A. Passi, C. Boggs, E. Smith, L. Gilham, T. Byun, and S. Rayadurgam. Run-Time Assurance for Learning-Based Aircraft Taxiing. In *Digitial Avionics Systems Conference (DASC)*, 2020.

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

[7] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, et al. Stable baselines, 2018.

[8] I. Hwang and C. E. Seah. Intent-based probabilistic conflict detection for the next generation air transportation system. In *Proceedings of the IEEE*, pages 2040–2059. IEEE, 2008.

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

[10] Kestrel Institute. ATJ: ACL2-To-C. https://www.kestrel.edu/research/atc/, 2022.

[11] Kestrel Institute. ATJ: ACL2-To-Java. https://www.kestrel.edu/research/atj/, 2022.

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

[13] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 1–18, New York, NY, USA, 2017. Association for Computing Machinery.

[14] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[15] J. Rivera, A. Danylyszyn, C. Weinstock, L. Sha, and M. Gagliardi. An architectural description of the simplex architecture. Technical Report CMU/SEI-96-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

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

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[18] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[19] 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, 2012.