Test-Case Generation for Runtime Analysis and Vice Versa: Verification of Aircraft Separation Assurance

Marko Dimjašević*
School of Computing, University of Utah, USA
marko@cs.utah.edu

Dimitra Giannakopoulou
NASA Ames Research Center, CA, USA
dimitra.giannakopoulou
@nasa.gov

ABSTRACT

This paper addresses the problem of specifying properties of aircraft separation assurance software, and verifying these properties at runtime. In particular, we target AutoResolver, a large, complex air-traffic control system that predicts and resolves aircraft loss of separation. In previous work, we developed a light-weight testing environment for AutoResolver. Our work contributed a wrapper around AutoResolver, which enabled the automated generation and fast execution of hundreds of thousands of tests. The focus of the work presented here is in specifying requirements for AutoResolver, in ensuring the generation of test cases that cover these requirements, and in developing a runtime infrastructure for automatically checking the requirements. Such infrastructure must be completely transparent to the AutoResolver code base. Our work combines test-case generation and runtime verification in innovative ways in order to address these challenges. The paper includes a detailed evaluation and discussion of our verification effort.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors, Testing tools*

General Terms

Reliability, Verification

Keywords

Aircraft separation assurance, aspect-oriented programming, runtime monitoring, test case generation, testing

*Author performed this work while employed by SGT Inc. as an intern at the NASA Ames Research Center.

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07...$15.00
http://dx.doi.org/10.1145/2771783.2771804

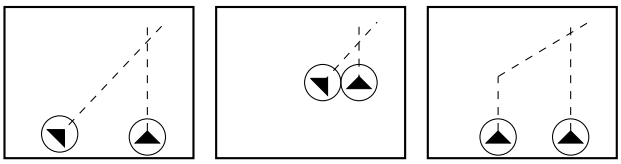

Figure 1: Loss of Separation and Resolution. Lateral view of two aircraft, their trajectories, and areas of horizontal separation assurance. Left: Conflict in the near future. Center: Loss of separation. Right: Detour that will prevent loss of separation.

1. INTRODUCTION

The Next Generation Air Transportation System (NextGen) is a NASA research program that addresses the increasing load on the air traffic control system through innovative algorithms and software systems. AutoResolver is a proposed NextGen component for prediction and resolution of loss of separation between multiple aircraft in the 1 to 8 minutes time horizon. Loss of separation between two aircraft occurs when they are closer to each other than a predefined safe vertical or horizontal distance. Separation assurance aims to eliminate the occurrence of loss of separation in the air space. Figure 1 shows a sketch of a potential loss of separation between two aircraft and how it can be avoided by letting one aircraft take a detour.

Testing AutoResolver presents various challenges. The input data consists of several aircraft trajectories, each trajectory being a sequence of 4-dimensional points, where a point represents a position in 3-dimensional space, plus a time instant. Given this complex input space of the separation assurance problem, it is extremely hard to generate appropriate input data for it. Therefore, the NextGen team typically uses historical airport data recordings as test inputs. Each such test case usually involves (tens of) thousands of aircraft and takes several hours to run. When unexpected behavior is detected, it is hard to create subsets of the test case that would lead AutoResolver to similar behavior, making debugging a complicated task.

To address these issues, the Robust Software Engineering (RSE) and NextGen groups at the NASA Ames Research Center have been collaborating over several years for the development of an automated, light-weight testing infrastructure for AutoResolver. In previous work [16] we developed a wrapper that implements parameterized loss of separation scenarios between two aircraft for AutoResolver. In contrast to having trajectories as inputs, which would make it

impossible for test-case generation tools to produce realistic trajectories, these scenarios expose parameters such as aircraft velocity and heading. These parameters provide flexibility in producing many different types of aircraft encounters for the problem, while always producing valid trajectory inputs that also exhibit loss of separation. Despite the fact that scenarios are limited to a single encounter between two aircraft, the wrapper has enabled us to experiment with both black-box and white-box test-case generation techniques, and to produce hundreds of thousands of tests aiming at exercising different aspects of the AutoResolver code.

A key aspect missing from our previous work is the identification of desirable properties for separation assurance software, and the support for automated testing of these properties. Introduced properties may, in turn, create additional requirements on the test-case generation itself: in order to exercise each property, we often need to produce complex encounter situations that target it. In general, it is hard to identify properties for separation assurance algorithms, as discussed in TSafe [15]. Currently, AutoResolver developers manually examine the outcomes of every test case to determine whether the software behaves as expected. This is impractical, more so in a setting where millions of test cases are generated and run automatically.

The work presented in this paper addresses these issues thus filling an important gap in our separation assurance testing framework. Specifically, it makes the following contributions:

- A set of requirements for separation assurance.
- Some of the requirements require test cases with multiple aircraft, including complex relationships between their trajectories. We therefore implement a generalization of the wrapper to support scenarios with any number of aircraft and loss of separation cases between them. Some aircraft are placed strategically in order to check specific properties of AutoResolver's logic.
- A runtime verification framework based on aspect-oriented programming for checking the requirements on AutoResolver. Runtime monitoring is also used to check property coverage by the generated tests, as well to monitor other behaviors of the system, as requested by its developers. The framework is completely separate from the AutoResolver code, thus allowing us to avoid interfering with the development process of the AutoResolver team.
- As well as being used for property monitoring, our runtime verification framework is used for complex test-case generation. This is, to our knowledge, a novel, atypical use of runtime verification.
- Application of our framework to AutoResolver and discussion of the obtained results.

The remainder of this paper is organized as follows. Section 2 provides background on AutoResolver and our previous testing framework for it. Separation assurance requirements for a system like AutoResolver are given in Section 3. Section 4 presents extensions to the interface and to the test-case generation capabilities of our testing framework, with Section 5 describing the runtime verification infrastructure that we develop on top of it. Evaluation results follow in Section 6, and lessons learned are provided in Section 7. Finally, related work and conclusions are discussed in Section 8, and Section 9, respectively.

2. BACKGROUND

This section provides a high-level description of AutoResolver and provides a brief overview of our previous work on developing a testing environment for it. Note that in the context of this work, we use the terms "conflict" and "loss of separation" interchangeably. Moreover, by the term "conflict time", or "*ttlos*", we refer to the first time point in their trajectories at which two aircraft lose separation.

2.1 AutoResolver

The Advanced Airspace Concept (AAC) is aimed at automating separation assurance in the future. AAC uses multiple independent layers of separation assurance for increased reliability. One component of AAC is a strategic problem-solving tool named AutoResolver [12]. AutoResolver's separation assurance algorithm was originally developed in the ACES simulation environment taking full advantage of the zero-error trajectory prediction available. Many studies of the effectiveness of this algorithm in the zero-uncertainty environment have been performed [13].

In each round of operation, AutoResolver attempts to resolve all the conflicts identified by its conflict-detection system, but handles them in the order specified by some notion of priority. The algorithm that it implements attempts to generate many different types of resolutions for each conflict. More specifically, AutoResolver iteratively attempts a variety of maneuvers, and determines which ones result in successful resolutions. Among all such resolution trajectories, AutoResolver selects the resolution that is expected to impart the minimum airborne delay.

AutoResolver consists of approximately $65K$ lines of Java code. For each maneuver that it attempts, it communicates with ACES, a simulation environment whose core consists of approximately $450K$ lines of code. As mentioned, the NextGen team typically uses historical airport data recordings as test inputs. These test cases consist of $4,800$ or $10,000$ aircraft, and take between 3 and 7 hours to run. The results are monitored manually by domain experts to see whether AutoResolver behaves as expected.

2.2 A Lightweight Testing Framework

To ensure more systematic testing of the behavior of AutoResolver, potentially targeting some type of test coverage, we previously developed a lighter-weight testing environment to complement the current testing process [16]. That testing environment has the following features. Despite the fact that ACES is very precise, it is a heavyweight tool that adds a significant burden to the testing process. We therefore created stubs that replace the functionality of ACES with more approximate behavior. The main capability that the ACES stubs provide is the generation and modification of aircraft trajectories. Stubbing ACES allows us to run tests significantly faster, which is important in a setting where millions of test cases are generated and executed automatically.

Moreover, to enable meaningful test-case generation, we created a modular, extensible wrapper around AutoResolver that implements parameterized conflict scenarios. Note that loss of separation is handled for two aircraft at a time (all other aircraft are called "secondary"). Each

proposed resolution is evaluated against the set of all aircraft in the airspace sector that AUTORESOLVER is currently handling.

More precisely, the wrapper provides an interface to AUTORESOLVER that consists of a number of entry points where each point represents a single-conflict scenario suggested by the AUTORESOLVER team. In a CRUISE scenario (Figure 2(a)) each aircraft is in level flight, i.e., its altitude remains constant. In a CLIMB scenario (Figure 2(b)) each aircraft climbs or descends for some portion of the trajectory. Loss of separation may occur before one or both of them start climbing or descending, during climb or descent, or after one or both of them has leveled off. Finally, the TURN scenario (Figure 2(c)) is similar to CRUISE, but it introduces a heading change for one or both aircraft at some point in their trajectory. Resolutions for the conflicts depicted in Figure 2 (a) – (c) are shown in Figure 2 (e) – (f), respectively. Each scenario is hard-coded in the type of trajectories, but is parameterized on aspects like aircraft velocity, initial heading, initial altitude, climb rate, and conflict time. Each concretized scenario is translated into a set of trajectories with which the wrapper invokes AUTORESOLVER.

In order to ensure that the test inputs that are thus generated mostly correspond to loss of separation scenarios, the framework works as follows. A point is selected in three-dimensional space, representing a position at which the two aircraft will meet (note that loss of separation actually occurs prior to the airplanes reaching that point). Aircraft are then flown backwards (headings are reversed) from their meeting point for the duration specified by the conflict time parameter. The type of trajectory is as specified by the scenario, i.e. whether the aircraft climb or cruise, for example, but specific details are parameterized, such as the time point at which a climb trajectory levels off. Aircraft initial positions are thus computed, and the ACES stub is then invoked to generate concrete trajectories.

The wrapper enables the application of a variety of test-case generation tools or algorithms for this problem, as demonstrated in our previous work. In this work, the focus is on an essential aspect of testing, which is how to evaluate the testing outcomes. In particular, we want to lighten the load of domain experts in monitoring each test case. Rather, our goal is to automatically check each test case against requirements, and focus the attention of the domain experts only on those tests that exhibit unexpected behavior.

3. SEPARATION ASSURANCE REQUIREMENTS

The first step in verifying the behavior of AUTORESOLVER is to create a specification, i.e. a set of requirements it should meet, which is not straightforward for a system that solves an optimization problem. Through several iterations of discussions with the AUTORESOLVER team, we have formed two types of requirements: verification properties and information monitors. Verification properties are statements about the expected behavior of AUTORESOLVER. They capture the high-level logic of the system, which conveys how the system operates in various situations. Information monitors provide a rich insight into specifics of the logic, which are not necessarily characterized as correct or incorrect.

3.1 Verification Properties

P_1 *There should be a resolution for every conflict.* The main goal of AUTORESOLVER is to predict and resolve conflicts. Therefore, it should be able to resolve every conflict that occurs within its time horizon.

P_2 *Initial conflicts are resolved in the non-decreasing order of their time to first loss of separation.* It is important that AUTORESOLVER handles conflicts with earlier conflict time first. Initial conflicts are those detected before a conflict resolution process starts. Conflicts that will happen in e.g. 7 minutes are not as urgent to resolve as those that will happen in 1 minute. In this way the conflict resolution process prioritizes to resolve more imminent conflicts first, and move to later conflicts according to their time to loss of separation.

P_3 *New conflicts arising as a result of conflict resolution should be inserted into the list of conflicts according to their time to first loss of separation.* As AUTORESOLVER is resolving initial conflicts, new conflicts can be created as a result of the resolution process. This means that a resolution trajectory of the maneuvered aircraft has a conflict with another aircraft, while its original trajectory did not have a conflict with the same aircraft. When AUTORESOLVER picks a resolution that results in a new conflict, the new conflict should be resolved according to its conflict time. This property is similar to property P_2, but instead it characterizes conflicts that did not exist originally.

P_4 *No picked resolution is allowed to cause a more imminent secondary conflict.* For every conflict AUTORESOLVER tries to resolve, it attempts multiple resolutions. Among the successful ones, it picks the best according to a set of optimization criteria. The picked resolution should not make the situation worse by getting the aircraft into a more imminent conflict than the conflict being resolved.

3.2 Information Monitors

The AUTORESOLVER team is interested in the stability of the picked resolution's type for a given conflict if the resolution process is delayed. Each resolution has a type, such as a horizontal maneuver, a temporary altitude change maneuver, etc. The team said that stability is important from the perspective of people working in air traffic control. In other words, the team wants to know if AUTORESOLVER would pick a different resolution type for the same conflict, if the resolution process was to be delayed for a given amount of time. We therefore formulate the following monitor:

M_1 *For each conflict, report its resolution type and how it changes over time.* We explain how we introduce a resolution delay in Section 4.

4. EXTENDING AUTORESOLVER'S TESTING FRAMEWORK

Our previous work [16] supported single-conflict pre-selected scenarios, in spite of AUTORESOLVER being able to resolve any number of arbitrary conflicts at a time. Our extensions to the previous work [16] are motivated by the need to support automated verification of separation assurance properties, and in particular the requirements presented in Section 3.

With this work we aim to verify properties of AUTORESOLVER's operational logic pertaining to more than one conflict. We therefore rewrite the framework interface to support the generation of any combination of any number of

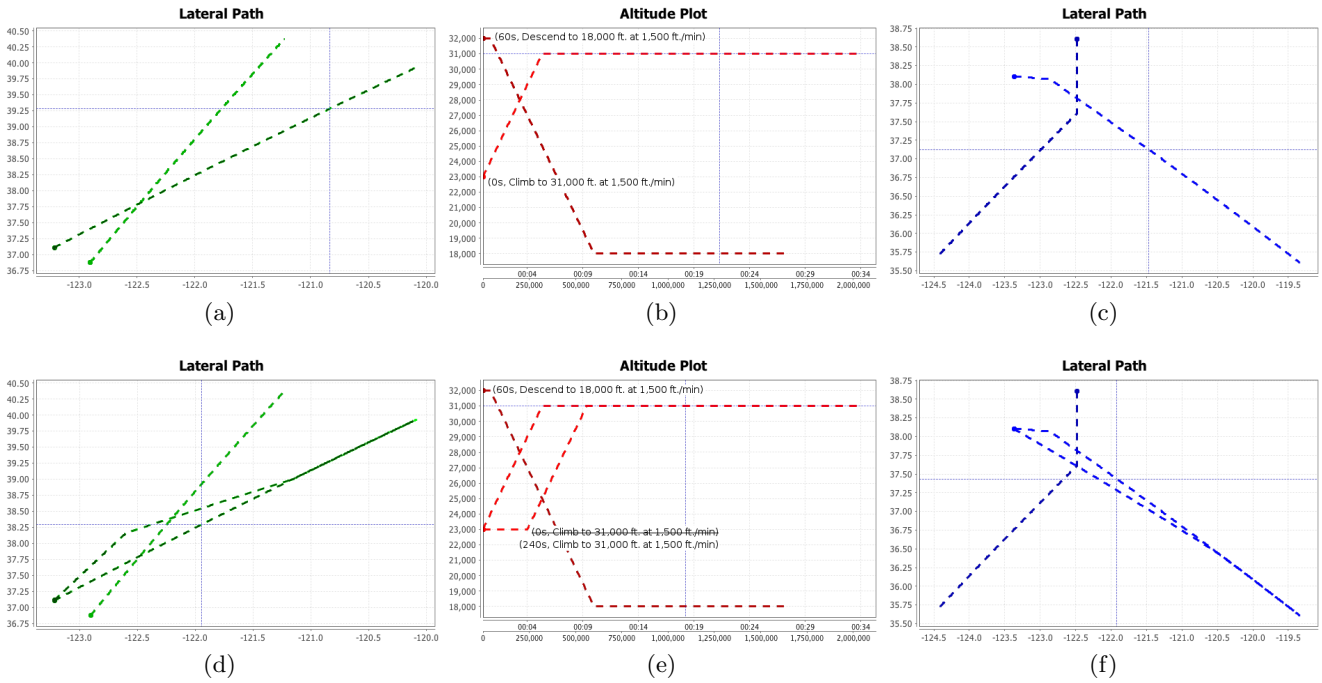

Figure 2: Conflict Scenarios and Examples of Successful Resolutions. The conflict and resolution visualizations are rendered with AacViz, a trajectory visualization tool developed by the AutoResolver team. (a) Lateral view of Cruise conflict and (d) Resolution through *Path Stretch* maneuver; (b) Longitudinal view of Climb conflict and (e) Resolution through *Extend Altitude* maneuver; (c) Lateral view of Turn conflict and (f) Resolution through *Direct To* maneuver. Axes of lateral plots are longitude and latitude in degrees. Axes of altitude plots are time (in milliseconds and minutes) and altitude in feet.

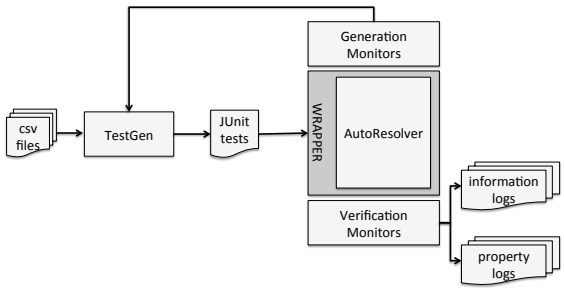

Figure 3: Overview of extended testing framework.

conflicting aircraft, including aircraft that create secondary conflicts (we call these secondary aircraft). An overview of the extended framework's architecture is provided in Figure 3.

4.1 Framework Interface

The new interface decouples the task of creating an aircraft trajectory from that of creating a conflict encounter. More specifically, a single aircraft can be added that flies a parameterized type of trajectory (e.g., cruise or climb) through an arbitrary 4-dimensional point [1]. In other words, the trajectory is such that the aircraft is at the 3-dimensional location specified by this point, at the time that is associated with the point. Two aircraft are put in conflict by making them fly through the same 4-dimensional conflict point, or have them be in a secondary conflict, as explained in Section 4.3. Unlike the previous framework, encounter scenarios are not hard-coded, but are generated through the combination of aircraft with appropriate trajectories. For example, a CLIMB conflict is generated by flying two climbing aircraft to the same conflict point.

```
public void test0() throws Throwable {
  AacTestWrapper wrapper = new AacTestWrapper();

  wrapper.setUpCR(CR_params1, conflict_point1);
  wrapper.setUpCRH(CRH_params1, conflict_point1);
  wrapper.setUpCL(CL_params1, conflict_point2);
  wrapper.setUpCL(CL_params2, conflict_point2);

  wrapper.runConflictDetectionResolution();
}
```

Listing 1: A test case with two pairs of aircraft in two independent conflicts.

A simplified example of a test case we generate according to the new interface is shown in Listing 1. It features two pairs of aircraft resulting in two independent conflicts. Each aircraft is added to the framework by utilizing a setup interface entry point that is specific to a particular aircraft trajectory type. (e.g. a CRUISE trajectory is generated by calling `setUpCR`.) Each of the setup methods takes a number of parameters defining the trajectory, and the targeted conflict point. When all aircraft are added to the framework, a wrapper call to AUTORESOLVER's detection and resolution process is made

[1]One of the parameters enables a time offset to the time dimension of the specified point

with the last statement in the test case.

A new addition to the interface is the ability to fly all aircraft for a given amount of time before invoking AutoResolver's conflict detection and resolution process. With this ability we stress-test AutoResolver's conflict detection and resolution of the same aircraft trajectories, but at different time points. At the same time, it allows us to observe how resolution types change over time. Once all aircraft are added to the framework, some of them exhibiting loss of separation, the framework invokes AutoResolver to detect and resolve conflicts between the aircraft.

4.2 Generating Multiple Conflicts

With the exception of P_1, all the properties reason about cases where AutoResolver handles multiple conflicts between more than two aircraft. In this work we introduce a way of generating test cases with multiple independent conflicts. This enables us to exercise the properties and report potential violations.

In our previous work, we generated test cases by computing the Cartesian product of all input parameters for an interface entry point describing a pre-selected single-conflict scenario. For the case of multiple conflicts, if we were to additionally compute the Cartesian product across conflict points, the resulting number of test cases would be prohibitively large. There are many ways in which we could handle this problem: one would be to implement a randomized approach to creating combinations of values; another would be to use combinatorial testing [7]. We plan on investigating the effectiveness of such techniques in the context of this system in the future.

As a first approach to the problem, we decided to introduce initial conflicts at points in the 3-dimensional space that are far enough from each other such that two aircraft involved in a conflict do not interfere with aircraft from the other conflicts. By making the conflicts isolated from each other, characteristics of one conflict are expected to be independent from characteristics of every other conflict. Based on this, it is sufficient to explore the Cartesian product of the parameter values for each conflict, but it is not necessary to also explore the Cartesian product across conflicts.

Each initial conflict is described by several parameters such as the time to the first loss of separation, aircraft trajectories, relative headings of the two aircraft, etc. Each conflict parameter is a floating point or an integer value. We range over its values by specifying an interval with a lower bound, an upper bound, and a step between neighboring values to be extracted from the interval. We draw parameter specifications from input files, one line specifying one parameter, as illustrated in the following:

```
ttLOS,           DOUBLE, "[400.0 to 540.0 step 20.0]"
relativeHeading, DOUBLE, "[20.0  to 180.0 step 20.0]"
airspeed,        DOUBLE, "[450.0 to 550.0 step 50.0]"
```

For each conflict, we then explore the full Cartesian product of the ranges of all the parameters involved. However, we do not explore all combinations of parameters across conflicts. Rather, we create test cases by picking one set of generated values for each conflict, and covering all possible combinations of parameter values for each conflict. During this process, we avoid having the same time to loss of separation among conflicts in each test case by using different offsets in the order in which their respective values are picked.

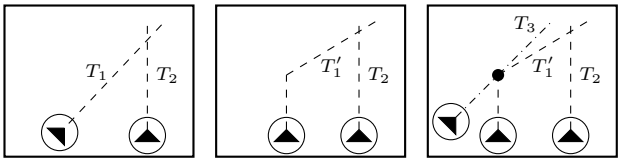

Figure 4: Loss of Separation Resolution and Introduced Secondary Conflict. Left: Conflict in the near future. Center: Resolution that will prevent loss of separation. Right: Secondary conflict introduced along the resolution trajectory.

4.3 Generating Secondary Conflicts

Properties P_3 and P_4 require for us to generate not simply additional aircraft, but also to strategically position them so that they generate secondary conflicts. In particular, we must create test cases that include secondary aircraft with which a resolved trajectory creates more and less imminent conflicts. If we were to take a random approach where we would include secondary aircraft without a specific goal, it would be extremely difficult to generate such test cases. We therefore decided to take control over positioning the secondary aircraft, in the same way that our wrapper takes control over generating initial conflicts.

The main challenge in generating secondary conflicts is the fact that we do not, apriori, know the resolution that AutoResolver will produce for a particular conflict. To deal with this problem, we develop the following approach, as illustrated in Figure 4. Let T_1 and T_2 be the trajectories of two aircraft in conflict, and assume that AutoResolver produces a resolution for T_1.

1. Monitor the behavior of AutoResolver and obtain the produced resolution. Generate a resolution trajectory T_1'.

2. Select a point in T_1' to which the secondary aircraft flies (black dot in Figure 4). Use our existing algorithms to fly the secondary aircraft backwards to the selected point and generate a trajectory T_3. The secondary aircraft will thus be in conflict with the resolution trajectory produced by AutoResolver.

3. Create a test case with T_1, T_2, T_3.

Similarly to test case generation of initial aircraft encounters, we want to parameterize the generation of secondary aircraft in order to be able to create interesting test cases. For example, we want to vary the time at which the secondary conflict occurs. For a desired time t to a secondary conflict, we pick an appropriate point in the resolution trajectory (the point at which the aircraft will be at time t), and create a secondary aircraft to reach that point also in time t, with the method discussed in Section 2.2. This enables us to create secondary conflicts that are both more and less imminent than the initial conflict. We also want to vary heading and velocity of the secondary aircraft, among other parameters. Finally, we need to make sure that when AutoResolver deals with the generated test case, it will still select T_1 and T_2 as the first conflict to resolve. In other words, if the secondary aircraft is in conflict with any of the initial aircraft, and this conflict is more imminent than the conflict between T_1 and T_2, then AutoResolver will

select the former conflict to resolver first, and therefore we no longer have control over the creation of the resolution trajectory that we create a conflict with.

To address this last issue, we use a runtime monitor which, prior to generating the test case including a secondary, checks whether the secondary aircraft has a conflict with the initial trajectories, and only generates a test case if the latter conflict is less imminent than the initial conflict. In other words, a test case such as above is only generated if the conflict between T_1 and T_2 is more imminent than both 1) a conflict between T_1 and T_3 if such a conflict exists, and 2) a conflict between T_2 and T_3, if it exists.

In summary, we use runtime monitors in order to query the behavior of AutoResolver and generate complex secondary scenarios in a strategic fashion. In this novel use of runtime monitoring for test-case generation, the software under test serves as a solver of our constraints for producing the test cases.

5. VERIFICATION AND MONITORING

A new component of the framework is developed to support verification and monitoring capabilities: monitoring the execution of AutoResolver, checking if properties hold, and recording property violations and other information to appropriate log files.

Log files for property violations let the AutoResolver team focus on those test cases that violate the properties, instead of manually examining results of millions of test cases. These log files provide information analogous to a regression test suite — if there is a test case violating any of the properties, a report will be written to the log files. Log files for the information monitor report detailed information on every aircraft conflict for every test case and how the picked conflict resolution's type changes over time.

5.1 Monitoring Information and Properties

One of the goals of this work is to keep AutoResolver's source code intact throughout the verification. What is usually done when verifying a software is to instrument its source code with in-lined commands needed for a verification task. The motivation behind avoiding such an invasive approach is the fact that we do not want to interfere with the ongoing development of AutoResolver.

Instead of modifying AutoResolver at the source code level, we add a verification component to our framework. The AutoResolver software is written in the Java programming language. Java compiles to Java bytecode, an intermediate language of the Java Virtual Machine. The verification component is written in the AspectJ language. AspectJ is an aspect-oriented programming extension to Java. It comes with a modified Java compiler that weaves the code of the component into AutoResolver's bytecode. This way we have AutoResolver's code interleaved with our verification code at the bytecode level only, thus leaving AutoResolver's source code unmodified.

For every property and the monitor introduced in Section 3, we have one aspect in the component. An aspect is an analog of a Java class. It consists of regular Java code in addition to pointcuts and advices. Pointcuts are moments in a program execution, e.g. an occurrence of a method call. Advices are actions to be taken before and/or after the pointcuts, e.g. fetching a return value after a method call.

In order to implement the runtime verification, we had to identify parts of AutoResolver's source code pertaining to the properties and monitor. Once the parts were identified, we were able to write aspects — consisting of pointcuts and advices — that implement the properties and monitor. The pointcuts also reach into the framework since the aspects initialize various parameters on the boundary from the framework to AutoResolver. Furthermore, the aspects reach into test cases exercising the framework and AutoResolver, again initializing parameters before a test case starts or outputting results after a test case execution or when a whole test suite ends with its execution. An example of a pointcut and an advice applied before the pointcut is shown in Listing 2.

```
pointcut executionJUnitTestMethod():
  execution(public void test*()) &&
  within(TestCase+) &&
  !cflow(myAspect());

before(): executionJUnitTestMethod() {
  currentTime = 0.0;
  resolutionInfoMap =
    new HashMap<AircraftIDPair, ArrayList<
        ResolutionInfo>>();
}
```

Listing 2: A pointcut and advice initializing parameters before a JUnit test case method execution.

Every aspect is instantiated at the beginning of a framework execution and it monitors an execution of each test case in a test suite, independently of other aspects. Some information the aspect observes and collects is written to log files after every test case, like the `resolutionInfoMap` data structure in Listing 2, while the rest of the information is written to the files only at the end of a test suite execution.

The resolution monitor modifies each JUnit test case and executes it for nine different time points. The resulting executions represent a class of test cases with the exact same aircraft trajectories. However, for each delay time, all aircraft involved in the test case are first flown for the specified amount of time prior to invoking the AutoResolver's conflict detection and resolution process. This is equivalent to generating and executing nine times as many test cases as in the original JUnit test suite. A pointcut and advice that achieve the runtime modification and execution are shown in Listing 3.

```
pointcut callAR(AacTestWrapper wrapper):
  call(public ArrayList
      runConflictDetectionResolution()) &&
  target(wrapper) &&
  !cflow(myAspect()) &&
  !cflow(callFlyForMethod(*, *)) &&
  if(isEnabled);

after(AacTestWrapper wrapper): callAR(wrapper) {
  for (t = 60.0; t <= 480.0; t += 60.0) {
    AacTestWrapper w = wrapper.flyFor(t);
    w.runConflictDetectionResolution();
  }
}
```

Listing 3: A pointcut and advice delaying the conflict detection and resolution process for every JUnit test case.

In testing AutoResolver, we make it easy to focus on specific properties or the runtime monitor through a configuration file. The file contains key-value pairs, where a

key represents one of the properties or the monitor, and a value is either *enabled* or *disabled*, allowing or disallowing the property or the monitor to be exercised, respectively. Note that when the information monitor is enabled, each test case results in multiple invocations to AutoResolver, one for each resolution delay that is introduced. In addition to that, all enabled property monitors are also active and checked as AutoResolver is invoked. For each test case, we therefore check all enabled properties at all resolution delay time points.

5.2 Monitoring the Monitors

In our previous work [16] we measured and compared coverage of our generated test suites in terms of both standard structural coverage criteria, but also in terms of the sets of attempted and successful resolutions.

When properties are introduced into the system, we have the opportunity to introduce additional coverage criteria for the generated test suites. In particular, we wish to make sure that our test suites include enough different scenarios to exercise the logic of all the properties that are to be checked in the system. One can view this as a type of "property coverage" criterion. If no test case exercises a particular property, our testing process satisfies this property "vacuously".

Let us consider property P_4, for example. This property requires that a picked resolution does not create a more imminent secondary conflict. One can imagine this property as having two branches, one for the case where a secondary conflict is created as a result of a resolution trajectory, and one where it is not. The first branch consists of two branches, one where the secondary conflict is more imminent than the original one, and one where it is less imminent. To cover this property, we therefore would like to have test cases that cover all three logical branches.

Similarly, property P_3 checks whether secondary conflicts are added to the list of conflicts in the correct order. In order to check the logic of this property, we must generate test cases that create secondary conflicts *within* the AutoResolver time horizon, so that they are eligible to be added to the list of conflicts.

In order to evaluate the quality of our generated test suites with respect to a set of properties, we introduce runtime monitors, which in essence observe whether the property monitors are exercised properly. This can be performed by extending the property monitor itself. For example, for property P_3, we extend the property monitor to additionally record the number of calls made to the method that adds secondary conflicts to an existing list of conflicts. Alternatively, it can be performed by introducing a new monitor. Since the life-span of aspects is through a test suite, it is easy to aggregate such test-suite coverage results.

6. VERIFICATION RESULTS

In this section we report results on verifying the properties and running the monitor introduced in Section 3. We analyze if each of the properties holds and what insights this gives us into how AutoResolver operates. For the monitor we look at the data it outputs and what it means.

6.1 Experimental Setup

Experiments were run on a 32-core computer with 128 GB of RAM in the Debian GNU/Linux operating system in the Emulab network testbed [32]. The experiments consist of generating test cases for AutoResolver, as described in Section 4.2 and then checking the properties and running the monitor while executing the test cases, as explained in Section 5. Every JUnit test case we generate consists of five independent initial conflicts, which equals to five pairs of aircraft, each pair flying to a conflict point. In addition, there is another aircraft in the test case intended to cause a secondary conflict if AutoResolver was to choose the same conflict resolution for one of the conflicts in the presence of the additional aircraft.

It takes 2.08 seconds on average to execute a test case. Given that it takes so long to execute a test case and that we generated 3.5 million test cases in total, we decided to implement the producer-consumer problem in the framework. The producer is the test case generator where each consumption unit is a batch of 5000 test cases. In the experiments we employed 30 consumers, each consumer executing one batch at a time. With the producer-consumer approach it took us about 3 days to run the test cases compared to about 84 days it would take us if we executed them sequentially.

One thing to note is that, as explained in Section 5, every test case is modified and executed in nine different ways at runtime. Effectively this means we executed not 3.5 million test cases, but 31.5 million modified test cases. Thanks to the runtime modifications of every test case representing multiple conflict scenarios, we were able to check all the properties for the same scenarios, but at different time points.

6.2 Property Checking / System Monitoring

In this section, we present the results we obtained from the runtime monitors that we implemented as oracles during the execution of our generated tests.

P_1 *There should be a resolution for every conflict.* Monitoring of this property identified several test cases for which AutoResolver was not able to produce a resolution. All these cases fall within four categories, as reported by the output of AutoResolver. Three categories have to do with the conflict time. More precisely, there are some conflicts that AutoResolver considers outside (before or after) its time horizon. For really imminent conflicts, tools such as TSafe [15] and TCAS [22] are in charge. For conflicts that are far enough in the future (how far in the future is configurable), it is often better to wait and see how they evolve before trying to resolve them. After all, each aircraft maneuver that is to be applied is disruptive in one way or another. There are also cases identified as "planes already in violation", which means that the initial states of the trajectories are already in loss of separation.

The fourth category produces a message that we could not directly interpret based on our prior experience with the tool: "Neither plane able to maneuver/neither plane able to be unfrozen". The AutoResolver developers informed us that this happens when aircraft involved in a conflict have already received a resolution in this round. By reviewing the logs from these test cases we confirmed that this was indeed the case. This is the type of behavior that we could not observe with the single-conflict test cases of our previous work.

The behavior of AutoResolver in all these cases is therefore as expected. Relating to the conflicts that fall outside of the AutoResolver time horizon, one could refine the property monitor to exclude such cases from being reported. Alternatively, in order to keep the property general, one could

simply create filters to be applied offline to the logged test cases. One could also create parameterized runtime monitors that allow to configure the time window within which the target separation assurance algorithm is expected to operate.

P_2 *Initial conflicts are resolved in the non-decreasing order of their first time to loss of separation.* We did not detect any violations to this property.

P_3 *New conflicts arising as a result of conflict resolution should be inserted into the list of conflicts according to their first loss of separation time.* We did not detect any violations to this property.

P_4 *No picked resolution is allowed to cause a more imminent secondary conflict.* Our framework initially reported several violations of this property, with the following message, for example:

Resolution for conflict: (ac1=1, ac2=2) ttlos = 75.0 [s], res type = 13 is causing a more imminent conflict with ac3=3 with ttlos = 0.0 [s]

We observed that in all the cases reported, the more imminent secondary conflict that occurs as a result of applying a resolution maneuver occurs immediately, with *ttlos* equal to 0.0. We performed several tests to confirm that our runtime monitor was detecting a real violation, and that our runtime monitor does not have a bug. In trying to figure out these violations with the AutoResolver team, we provided to them detailed trajectories involved in one of these test cases. One of the developers observed the fact that in our test cases, the initial point of the original trajectory did not coincide with the initial point of its corresponding resolution trajectory.

This fact exposed a bug in our wrapper, which we had not discovered previously. In debugging the issue, we noticed that when our ACES stub creates a trajectory, the first point that it adds to the trajectory is the first point to which the aircraft flies. In other words, all the trajectories generated by our stub are offset by 5 seconds (trajectory points are 5 seconds away from each other). This is not a serious problem since it does not really matter what the initial point of an aircraft is, for the purpose of our testing. However, when the ACES stub is asked to create a trajectory that starts at a particular initial point I, it is reasonable to expect that the first trajectory point will coincide with I.

Despite this fact, we were surprised that AutoResolver produced a resolution with a more imminent secondary conflict. We were expecting that when a resolution is attempted, AutoResolver would check whether it creates more imminent conflicts. Since the conflict detection algorithm detects such a conflict, how is it possible that AutoResolver selects the resolution? After several interactions with the AutoResolver developers on this issue, we realized that the logic of the algorithms used assumes that the initial point of original and resolution trajectories are the same. When we updated the ACES stub, the violations to this property disappeared.

This attests to the correctness of our runtime monitor. Runtime verification thus proves invaluable in debugging and understanding a system under test, which is impossible without the use of test oracles to assist in this process. We identified an issue with our ACES stub, but also discovered an implicit assumption made by the logic of AutoResolver. We believe that this feedback is useful to the AutoResolver team. An assumption is made that the trajectory generator will always create trajectories with a specific initial point, but why not robustify the logic to work correctly if a trajectory generator does not exactly satisfy this criterion?

M_1 *For each conflict, report its resolution type and how it changes over time.* This monitor produces, for each conflict of each test case, a report that looks as in Table 1. This report allows for an easy inspection of how the produced resolution type changes when conflict resolution gets postponed. Visual inspection is of course not the aim of our work. Monitor M_1 produces about 20GB of logged data. This information opens up many opportunities for data analysis. One could, for example, try to calculate the average, or minimum, or maximum delay at which the produced resolution type changes. One could also look for characteristics of outlier cases. Through observation of the reports, for example, we have noticed some cases that look irregular; for example, it is sometimes the case that a conflict ahead disappears and then reappears between two aircraft, which is not very intuitive. In other cases, the resolution type changes very early, where in most cases the resolution type only changes when we delay by at least a couple of minutes.

Table 1: Resolution delay and resulting resolution types. Non-resolved conflict 1 is reported as being "before AutoResolver horizon", and 2 reports "planes already in violation". Resolution types are represented as integers in AutoResolver.

ttlos [s]	Delay time [s]	Res type
430.0	0.0	26
370.0	60.0	26
310.0	120.0	26
250.0	180.0	26
190.0	240.0	26
130.0	300.0	13
70.0	360.0	13
10.0	420.0	not resolved[1]
0.0	480.0	not resolved[2]

We plan to work with the AutoResolver team in order to develop offline monitors that process logged data to facilitate the interpretation of such results. We also plan, in the future, to apply data analysis techniques that go beyond the scope of this work.

Property Coverage. As discussed, in addition to checking properties, we have introduced monitors that check whether properties are exercised appropriately by the executed test suite. Our test suite exercises all properties appropriately except for property P_3. We were puzzled by this behavior. For this reason, we additionally monitored whether we generate test cases with the following characteristic: a resolution is picked that introduces a less imminent conflict than the original one, and the secondary conflict is within the AutoResolver horizon. After confirming this fact, we had extensive discussions with the AutoResolver team, and discovered that for the types of conflicts that we are implementing (called en-route), secondary conflicts are not handled in the current iteration of AutoResolver. Rather, they are left to be detected in the next invocation of the tool. Only weather conflicts, which we have not yet incorporated in our framework, may exercise this behavior.

7. LESSONS LEARNED

Testing without oracles is a shot in the dark. We thought that through the years of trying to understand how to generate test cases and stubs for a complex system like AutoResolver we had learned many aspects of its behavior. However, the pace at which we discover new aspects of its logic has increased significantly with the introduction of runtime monitors. Monitoring the system has helped us identify assumptions of the AutoResolver logic, categories of unresolved conflicts that we were unaware of, as well as bugs in our own wrapper code (after all, the wrapper is a part of our system under test) that we had not previously discovered despite extensive testing [16]. And this is without counting all the information that we expect to learn from the recorded data by the information monitor.

Strategic test-case generation and property monitoring are valuable tools in the hands of developers. In presenting our results to the wider separation assurance team, there were two things that sparked their enthusiasm. The first was the novel way in which we generate secondary aircraft. Of our other test-case generation work they said that "we have done somewhat similar work for our own basic testing, although not nearly as advanced or flexible". However, they said that "we have never seen anything remotely similar to this way of generating secondary conflicts; it is very novel and interesting".

The second aspect of our work that they believe can make a real impact in the way they test their software is the runtime monitoring. They believe that verification through monitoring can significantly facilitate their testing and regression processes. They also appreciate the fact that monitors do not interfere with their development.

Properties trigger the creation of new properties. From this, but also our past experience with TSafe [15], it is clear that the best way to identify properties of complicated algorithms, is to start from simple ones, and show the value to system developers. The capability to create logs of tests that identify several characteristics of interest immediately creates a desire in the team to monitor additional aspects of the system. This is a great opportunity for formal methods experts to get their hands on real systems. Of course, it comes at the price of a significant engineering effort, which, in our case, has spanned over several years. But the potential impact of our work makes the effort worthwhile and rewarding.

Our information monitor also opens up opportunities for applying a variety of data analysis and mining techniques in this domain. One example of properties that we have not yet explored are properties that compare resolutions across consecutive rounds of operation of AutoResolver. The difference from our current information monitor is that instead of flying aircraft in order to delay the resolution process, we would actually apply the picked resolutions, and invoke AutoResolver again on the resolved trajectories, at the next time point according to the frequency of invocation. We are inspired to specify such properties from work on analyzing the ACAS X system [31]. In that work, an example of undesirable behavior is described as "reversals", which describes a situation where the same aircraft is advised to climb and subsequently to descend. Such maneuvers are extremely disruptive to the pilot.

Runtime monitoring for test-case generation. In this work, we use runtime monitoring in conventional ways, to check properties of the system under test, but also in innovative ways, in order to generate test-cases that are hard to generate with other techniques. In particular, there are two such interesting examples. The first one has to do with creating secondary conflicts. In this example, the system under test is used as a "constraint solver"; it informs our test-case generation tool of the resolution that AutoResolver would pick for a specific conflict. We believe that such an approach could be applicable in many real world scenarios where we are required to solve constraints that are very particular to a specific application.

The second case is where the runtime monitor captures information from one test case, modifies it, and subsequently runs the modified test case, as performed by our information monitor. M_1 modifies each test case at runtime to effectively generate a whole class of related test cases with the same aircraft setup, but across time.

8. RELATED WORK

In the research field of runtime verification researchers have explored a number of formalisms, usually with a trade-off between expressiveness and performance. Java PathExplorer [20] checks an execution of a Java program against user-provided properties and analyzes the program for deadlocks and data races. LogFire [19] is a rule-based runtime verification system based on a pattern-matching algorithm for implementing production rule systems. JavaMOP [5] enables the user to specify properties using a formalism, and automatically generates monitors in AspectJ from the specification. Even though we could have used a tool such as JavaMOP in this work, we did not need its expressiveness, hence we encoded the requirements in AspectJ directly. For an overview of the runtime verification field, see Leucker and Schallhart [23]. Runtime verification is also used in software-fault monitoring; Delgado et al. [10] provide a taxonomy and a classification of software-fault monitoring systems.

A combination of test-case generation and runtime verification has been explored by others working on runtime verification. The jUnit^{RV} tool [9] extends the JUnit unit testing framework with annotations that are generated from a user-specified temporal logic formula. The tool manipulates Java bytecode at runtime. Artho et al. [2] use an input-output model to automatically generate test cases enriched with verification properties. The test cases are generated using symbolic execution and the properties are analyzed by applying runtime verification to execution traces. In their approach, the system under test is instrumented manually so that events of interest are recorded in the execution traces. In our work, runtime verification is instrumental in test-case generation, and instrumentation is done automatically at the Java bytecode level, without affecting the source code of the software under test. Furthermore, in our work, runtime monitors and test cases are separate, offering the flexibility to apply any selection of runtime monitors to any selection of test cases (whether generated automatically or manually).

The AutoResolver system has previously been integrated and evaluated with other National Airspace System (NAS) simulations [24, 27, 28]. Moreover, in previous work, we tested TSafe [15], a NextGen component that also targets separation assurance, but at a shorter time horizon. That work also identifies properties to be tested, but these properties involve the input and output data of the system, and therefore do not require more involved monitoring, as in this work. Moreover, the input space of TSafe is much smaller

than that of AutoResolver, making test-case generation simpler.

Several researchers have addressed the challenge of generating structurally complex inputs with white- and black-box techniques. Techniques range from using declarative specifications of the test inputs [4, 17] to white-box techniques based on concolic execution for security testing [18]. MACE [6] combines black- and white-box techniques such as active automata learning and concolic execution in order to increase code coverage. A lot of research combines the techniques to generate method sequences and input values for primitive parameter types and objects [30, 21, 29, 11, 25, 14]. Arnold and Alexander [1] generate complex tests based on an automated approach to generating content for computer games. Other researchers rely on program invariants inferred from executions in generating test cases [3, 8].

As mentioned, other test-case generation approaches can be added to our framework such as plain random input generation, concolic execution, combinatorial testing, and evolutionary test case generation [26]. The use of such techniques is only made possible by our implementation of a wrapper to tame the input space of AutoResolver.

9. CONCLUSIONS

Over several years, we have been working with experts in aircraft separation assurance to develop a light-weight testing environment for the AutoResolver tool. In this work, we particularly focus on specifying and monitoring properties of the AutoResolver algorithms during testing. We have discussed how we implemented property and information monitors in AspectJ, allowing us to log several aspects of the system without interfering with its source code.

In order to effectively exercise properties of interest of the system, we had to generate sophisticated test cases. To this aim, we used runtime monitoring in innovative ways, both as an oracle for constraint solving, and as a generator of classes of related test cases.

Note that in this paper we do not focus on experimenting with different test-case generation tools. Rather, we focus on creating appropriate interfaces around AutoResolver, which tame its input space of trajectories into parameterized scenarios that can be handled by test-case generation tools. Although in this work we have used a simple black-box test-case generation algorithm, we will experiment, in the future, with using alternative techniques such as: randomized or combinatorial testing, concolic execution, or learning-based testing approaches.

We have automatically generated and efficiently executed millions of test cases, and discovered errors and vulnerabilities in the system consisting of AutoResolver and our wrapper code. We also gained new insights in the logic of the algorithms. The separation assurance team at NASA Ames has expressed interest in incorporating our work more widely. We are also interested in creating generic monitor templates for separation assurance, that can be configured to work with a variety of tools and algorithms in that domain.

Finally, the AutoResolver team would like us to stress-test the system by placing conflict points closer together, for example, and thus having aircraft from different conflicts interfere with each other. They are also interested in classifying input tests for which the AutoResolver may not be able to produce a resolution. In general, the information we produce opens many opportunities for further analysis.

Even though we have so far focused our work on air-traffic control algorithms, the approaches that we have developed are relevant in other domains. For example, we are currently collaborating with the car industry to test algorithms for self-driving cars.

10. ACKNOWLEDGMENTS

The authors wish to thank Heinz Erzberger, Todd Lauderdale, and David Thipphavong from the AutoResolver team for invaluable help with development of requirements, and understanding of the monitoring results. They also gratefully acknowledge Ivica Dimjašević, Falk Howar, and Raimondas Sasnauskas for proofreading the paper.

11. REFERENCES

[1] J. Arnold and R. Alexander. Testing autonomous robot control software using procedural content generation. In *SAFECOMP*, pages 33–44. Springer-Verlag, 2013.

[2] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Roşu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, pages 209–234, 2005.

[3] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 169–180, 2006.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.

[5] F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. In *Proceedings of the 22nd Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, pages 569–588, 2007.

[6] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. J. Wu, and D. Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the USENIX Security Symposium*, 2011.

[7] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Softw.*, 13(5):83–88, 1996.

[8] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology*, pages 1–37, 2008.

[9] N. Decker, M. Leucker, and D. Thoma. jUnit^{RV} — Adding runtime verification to jUnit. *NASA Formal Methods*, pages 459–464, 2013.

[10] N. Delgado, A. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, pages 859–872, 2004.

[11] M. Dimjašević and Z. Rakamarić. JPF-Doop: Combining concolic and random testing for Java. In *Java Pathfinder Workshop*, 2013. Extended abstract.

[12] H. Erzberger, T. A. Lauderdale, and Y.-C. Chu. Automated conflict resolution, arrival management and

weather avoidance for ATM. In *Proceedings of the 27th International Congress of the Aeronautical Sciences*, 2010.

[13] T. C. Farley and H. Erzberger. Fast-time simulation evaluation of a conflict resolution algorithm under high air traffic demand. In *Proceedings of the 7th USA/Europe Air Traffic Management R&D Seminar*, 2007.

[14] P. Garg, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 132–141, 2013.

[15] D. Giannakopoulou, D. H. Bushnell, J. Schumann, H. Erzberger, and K. Heere. Formal testing for separation assurance. *Annals of Mathematics and Artificial Intelligence*, 63(1):5–30, 2011.

[16] D. Giannakopoulou, F. Howar, M. Isberner, T. Lauderdale, Z. Rakamarić, and V. Raman. Taming test inputs for separation assurance. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 373–384, 2014.

[17] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 225–234, 2010.

[18] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, 2012.

[19] K. Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, pages 1–28, 2014.

[20] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, pages 189–215, 2004.

[21] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 297–306, 2008.

[22] J. Kuchar and A. C. Drumm. The traffic alert and collision avoidance system. *Lincoln Laboratory Journal*, 16(2):277, 2007.

[23] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, pages 293–303, 2009.

[24] D. McNally and D. Thipphavong. Automated separation assurance in the presence of uncertainty. In *Proceedings of the 26th International Congress of the Aeronautical Sciences*, 2008.

[25] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[26] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[27] T. Prevot, J. Homola, J. Mercer, M. Mainini, and C. Cabrall. Initial evaluation of air/ground operations with ground-based automated separation assurance. In *Proceedings of the 8th USA/Europe Air Traffic Management R&D Seminar*, 2009.

[28] D. Thipphavong. Analysis of climb trajectory modeling for separation assurance automation. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, 2008.

[29] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. *SIGPLAN Notices*, 46(10):189–206, 2011.

[30] N. Tillmann and J. d. Halleux. Pex—white box test generation for .NET. In *Proceedings of the International Conference on Tests and Proofs (TAP)*, pages 134–153, 2008.

[31] C. von Essen and D. Giannakopoulou. Analyzing the next generation airborne collision avoidance system. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 620–635, 2014.

[32] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, 2002.