Taming Test Inputs for Separation Assurance

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

Falk Howar
Carnegie Mellon University,
Moffett Field, CA, USA
howar@cmu.edu

Malte Isberner*
TU Dortmund University,
Dortmund, Germany
malte.isberner@udo.edu

Todd Lauderdale
NASA Ames Research Center,
Moffett Field, CA, USA
todd.a.lauderdale
@nasa.gov

Zvonimir Rakamarić*
School of Computing,
University of Utah, USA
zvonimir@cs.utah.edu

Vishwanath Raman*
FireEye Inc., USA
vishwa.raman@gmail.com

ABSTRACT

The Next Generation Air Transportation System (NextGen) advocates the use of innovative algorithms and software to address the increasing load on air-traffic control. AutoResolver [12] is a large, complex NextGen component that provides separation assurance between multiple airplanes up to 20 minutes ahead of time. Our work targets the development of a light-weight, automated testing environment for AutoResolver. The input space of AutoResolver consists of airplane trajectories, each trajectory being a sequence of hundreds of points in the three-dimensional space. Generating meaningful test cases for AutoResolver that cover its behavioral space to a satisfactory degree is a major challenge. We discuss how we tamed this input space to make it amenable to test case generation techniques, as well as how we developed and validated an extensible testing environment around AutoResolver.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Verification, Experimentation

Keywords

Test case generation, air-traffic control, test coverage

*These authors did this work while at Carnegie Mellon University.

(c) 2014 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the United States Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ASE '14, September 15 - 19 2014, Vasteras, Sweden
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2578726.2578744.

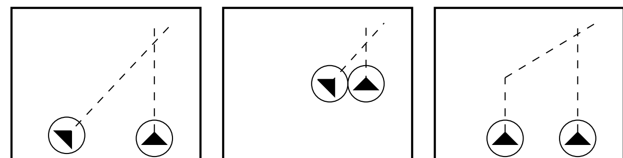

Figure 1: Loss of Separation and Resolution. Lateral view of two airplanes, their trajectories, and areas of horizontal separation assurance. Left: Conflict in the near future. Center: Loss of separation. Right: Detour that will prevent loss of separation.

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. This paper reports on collaborative work between the Robust Software Engineering (RSE) group and the NextGen group at the NASA Ames Research Center. The work aims at developing an automated, light-weight testing infrastructure for AutoResolver, a proposed NextGen component for prediction and resolution of loss of separation between multiple airplanes in the 1 to 20 minutes time horizon. Loss of separation between two airplanes 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 aircrafts and how it can be avoided by letting one airplane take a detour.

Testing AutoResolver presents various challenges. The input data consists of several airplane trajectories, each trajectory being a sequence of points in the three-dimensional space that the airplane is expected to fly. If trajectory points were exposed directly as input parameters to the testing problem, the vast majority of generated test cases would not correspond to a realistic airplane trajectory. It would also be extremely hard to express constraints between trajectory points to avoid this problem. Moreover, the AutoResolver logic involves millions of paths that must be tested thoroughly. A practical challenge when testing Au-

toResolver is that it was designed and developed as a component of a heavyweight airspace simulation environment named ACES [15].

Given the complexity of generating appropriate input data for AutoResolver, the NextGen team typically uses historical airport data recordings as test inputs. Such inputs usually involve thousands of airplanes, and each test case takes several hours to run. Note that it is hard to isolate subsets of the airplanes that would lead AutoResolver to similar behavior. Our efforts therefore concentrate on generating a light-weight but accurate enough testing environment, where thousands of test cases can be run within minutes to achieve some targeted coverage of AutoResolver. To achieve this goal, our team has developed the following components:

1. A wrapper that provides parameterized loss of separation scenarios for AutoResolver is presented in Section 3. Each concretized scenario corresponds to input airplane trajectories for AutoResolver. While allowing flexibility through its input parameters, this wrapper only generates realistic trajectories.
2. Stubs for the ACES database, which result in a less precise, but significantly more light-weight testing environment overall (Section 3).
3. Black-box and white-box techniques for test case generation at the wrapper level, as described in Section 4.
4. Tools for evaluation of our developed environment. We developed extensions to the coverage measuring tool JaCoCo[1] for comparing how different scenarios and test case generation techniques contribute to test coverage of AutoResolver (Section 5). We also connected our tools to NextGen visualization tools and evaluated the quality of the test inputs and the ACES stubs that we created (Section 6).

In previous work, we focused on TSafe [16], a NextGen component that also targets separation assurance but at a shorter time horizon. Testing the more complex AutoResolver system presented us with new challenges. TSafe is a standalone application whose inputs consist of position, airspeed, and heading for airplanes. This enabled us to generate inputs for the system directly; even so, only black-box test case generation was successful at scaling for TSafe. On the other hand, trajectory generation has been a major challenge in testing AutoResolver, as has been the stubbing of the ACES system. Moreover, in the last four years, we have developed robust white-box techniques; these have been applied successfully to AutoResolver, as reported in this paper. Finally, our new testing environment has been integrated with NextGen tools to make its application and evaluation simple and intuitive for NextGen developers.

Note that the topic of defining oracles for the testing process, although crucial for automated testing, is not addressed here. Similarly to TSafe [16], oracles are hard to define for this type of problem. For example, AutoResolver producing a successful resolution does not necessarily correspond to correct behavior, as the following requirement should be met: "a proposed resolution should not create a more imminent conflict with a third aircraft". We intend to identify and formalize oracles in the future, in collaboration with the domain experts.

[1] http://www.eclemma.org/jacoco/

2. AUTORESOLVER

The Advanced Airspace Concept (AAC) is a concept for automating separation assurance in the future. A key feature of this concept is the use of multiple independent layers of separation assurance for increased reliability. One component of AAC is a strategic problem-solving tool known as AutoResolver [12]. This algorithm was originally developed in the ACES environment taking full advantage of the zero-error trajectory prediction available, and many studies of the effectiveness of this algorithm in the zero-uncertainty environment have been performed [13].

AutoResolver uses an iterative approach to resolve all of the conflicts found by the conflict detection system. The algorithm attempts to generate many different types of resolutions for each conflict. After the resolution trajectories have been generated, the successful resolution expected to impart the minimum airborne delay is chosen for implementation.

AutoResolver consists of approximately $65K$ lines of Java code. For each resolution that it attempts, it communicates with ACES, a simulation environment whose core consists of approximately $450K$ lines of code. Testing of AutoResolver is typically performed using test cases of $4,800$ or $10,000$ airplanes, and take between 3 and 7 hours to run. The results are monitored by domain experts to see whether AutoResolver behaves as expected.

To ensure more systematic testing of the behavior of AutoResolver, potentially targeting some type of test coverage, a lighter-weight testing environment would be desirable to complement the current testing process. The capability to generate tests automatically would also be useful for generating smaller (in terms of numbers of airplanes involved), more focused, and easier to run tests. The results of our collaborative work with the AutoResolver team in creating such an environment are presented in the rest of this paper.

3. TEST ENVIRONMENT

As mentioned, the input to AutoResolver consists of a set of trajectories as sequences of 300 points that an airplane is expected to fly. Neighboring points are 5 seconds apart, and each one describes the three-dimensional position of the airplane (latitude, longitude, and altitude), as well as other details such as airplane speed and heading. Based on these trajectories, AutoResolver checks if loss of separation occurs between any pair of airplanes within its time horizon of 1 to 20 minutes. For any such pair, AutoResolver attempts to resolve the loss of separation by proposing maneuvers that modify the originally planned trajectory. The ACES tool is responsible for creating new trajectories for the proposed maneuvers.

3.1 Replacing ACES

ACES [15], developed at NASA Ames, is a sophisticated tool that simulates airplane flights taking into account airplane characteristics as well as other factors that may affect flight such as winds and weather. AutoResolver relies on the ACES trajectory generation in order to evaluate resolution maneuvers for each loss of separation scenario (as). Despite the fact that ACES is very precise, it is a heavy-

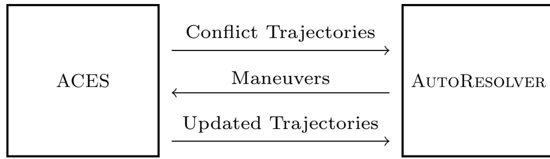

Figure 2: Integration of AutoResolver and ACES.

weight tool that adds a significant burden to the testing process. In creating a light-weight testing environment, we have therefore created stubs that replace the functionality of ACES with more approximate behavior.

The main capability that our ACES stubs provide is the generation and modification of airplane trajectories. In our implementation, each airplane is associated with an abstract trajectory. An abstract trajectory basically captures the intent of a trajectory; it consists of a number of commands that describe points in time at which the airplane changes airspeed, heading, maintains some temporary altitude, or climbs or descends to a target altitude. An abstract trajectory with no commands corresponds to simply cruising at the initial altitude with no heading or airspeed changes. In our framework, the AutoResolver maneuvers are then simply translated to the addition, removal, or modification of commands in the abstract trajectory.

Consider, for example, the trajectories illustrated in Figure 3 (b) and (e). The resolution displayed in (e) corresponds to an *Extend Altitude* maneuver, which delays a climb or descend command for a short period of time.

Abstract trajectories are turned into concrete trajectories to be understood by AutoResolver. This is achieved by computing flight points at 5 second intervals that correspond to the abstract trajectory and initial airplane position. We use the NASA open source WorldWind libraries [4] to perform required great circle calculations for computing latitude and longitude.

3.2 Scenarios

How does one automatically generate airplane trajectories to exercise the behavior of AutoResolver? Our goal with this work was to enable the application of existing test case generation approaches to this complex system. As mentioned, it is practically impossible to automatically generate realistic trajectories as sequences of unconstrained points in the three-dimensional space.

Imagine, for example, that one was to use some sophisticated symbolic execution tool [25, 30] to generate trajectories that exercise the paths of the AutoResolver code. Even if such a tool were able to scale to the size of the problem, the points generated by the tool would most likely not correspond to a trajectory that can be flown by an airplane. It would also be infeasible to introduce constraints between the points to ensure that they represent a viable trajectory. Moreover, our goal is for the majority of generated tests to represent scenarios where loss of separation occurs.

Our approach to dealing with this problem is to create a modular, extensible wrapper around AutoResolver that implements parameterized loss of separation scenarios. Each concretized scenario is translated into a set of trajectories with which the wrapper invokes AutoResolver. Note that loss of separation is handled for two airplanes at a time, while each proposed resolution is evaluated against the set of all airplanes in the airspace sector that AutoResolver is currently handling (all other airplanes are called "secondary"). We therefore create loss of separation scenarios between two airplanes.

In order to ensure that the test inputs that are thus generated mostly correspond to loss of separation scenarios, we proceed as follows. First of all, a point is selected in three-dimensional space, representing a position at which the two airplanes will meet. Even though loss of separation will actually occur prior to the airplanes reaching that point, for simplicity we will refer to this point as the loss of separation point. Given airplane heading, airspeed, altitude, an abstract trajectory representing the targeted scenario, and some time point t in the future at which the airplanes are to reach the loss of separation point, we fly the two airplanes backwards (headings are reversed) to reach their respective initial positions. Trajectory generation is then used, as implemented for the ACES stub, to create appropriate trajectories for the two airplanes.

In the following, we describe three types of scenarios that are currently implemented in our framework. These scenarios are well understood by the AutoResolver developers team to be standard loss of separation scenarios that may exercise the behavior of AutoResolver in interesting ways.

The Cruise scenario represents the simplest case where each airplane is in level flight (i.e., its altitude remains constant). An example of such a scenario is illustrated in Figure 3(a). Since the exact position at which each scenario occurs is not important, all parameters of one airplane are kept constant, as is the point of loss of separation (its lateral coordinates and altitude). However, we parameterize the scenario on the *heading (deg.)* and *airspeed (Kts.)* of the other airplane, as well as on the *time (sec.)* of loss of separation and the *offset (sec.)* with which the airplanes arrive at the specified point. By letting two airplanes pass the same point in space with some time in between, we can control if there will be a loss of separation, and create situations in which it is sufficient to slow down one airplane for a little while in order to avoid a conflict.

In the Climb scenario, both airplanes climb or descend for some portion of the short term trajectory. Figure 3(b) illustrates such an example, where one airplane descends and the other airplane climbs, and loss of separation occurs during descent/climb. Variations of this scenario include cases where the airplanes lose separation after one or both of them level off. In addition to the input parameters of the Cruise scenario, this scenario parameterizes the type of trajectory for each airplane through the specification of *initial cruise (sec.)* time, and the *level-off (sec.)* time, as well as the *climb (ft./min)* rates.

The Turn scenario is similar to Cruise but introduces a change of heading for one or both airplanes at some point in the trajectory, as illustrated in Figure 3(c). In addition to the input parameters of the Cruise scenario, this scenario is for each airplane parameterized on the *change of heading (deg.)* and the respective *time of change (sec.)*.

We would like to stress that these are a subset of the scenarios that one could define for this problem, and that we are currently only dealing with loss of separation during flight, as opposed to during arrivals and departures. We are also not dealing with weather conflicts. The architecture of our tool is extensible, making it easy for us, or the Au-

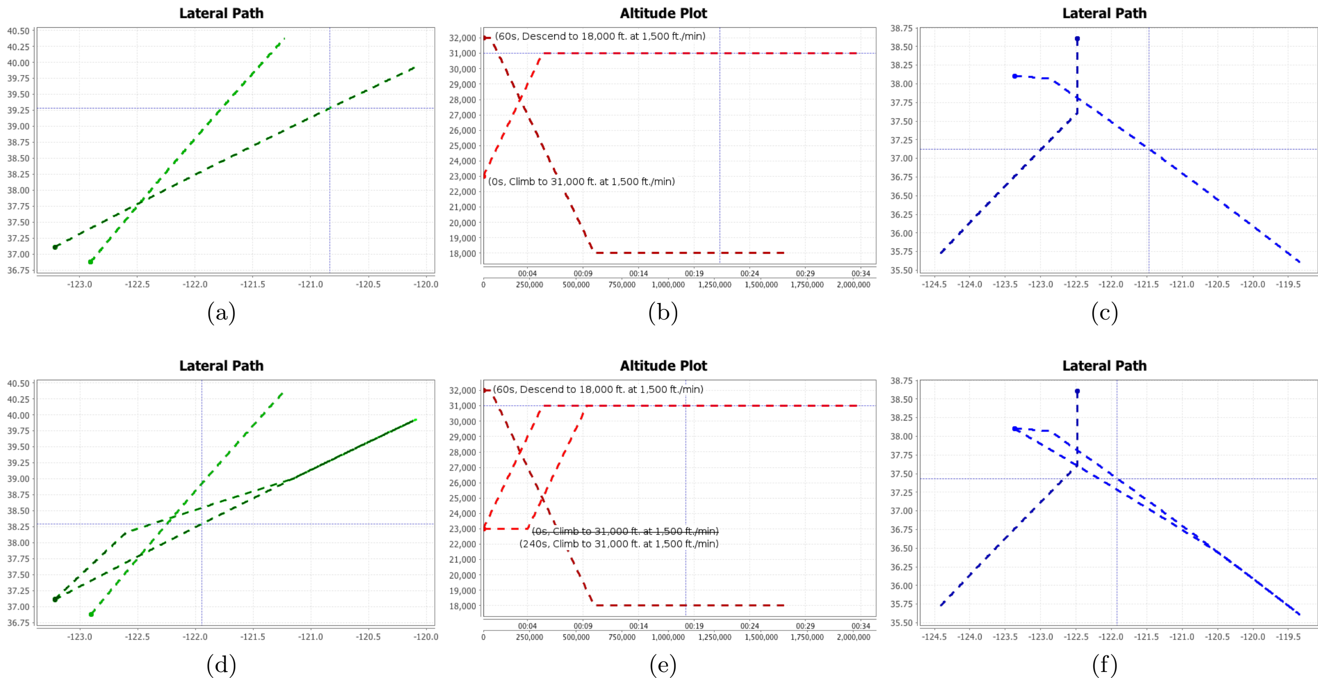

Figure 3: Conflict Scenarios and Examples of Successful Resolutions. (a) Lateral view of Cruise conflict and (d) resolution through *Path Stretch* maneuver; (b) longitudinal view of Climb conflict and (e) resolution through *Extend Altitude* maneuver; (c) lateral view of Turn conflict and (f) resolution 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. Fig. (b) and (e) show commands from abstract trajectories.

```
AacTestWrapper,testCLCL,((climb1 != 0.0 || climb2 != ...
  head1,     DOUBLE, "[    0.0 to  180.0 step   10.0]"
  ...
  climb1,    DOUBLE, "[-1500.0 to 1500.0 step 1500.0]"
  cruise1,   DOUBLE, "[    0.0 to  240.0 step  120.0]"
  levelOff1, DOUBLE, "[  360.0 to  500.0 step  120.0]"
  time,      DOUBLE, "[  120.0 to  540.0 step   60.0]"
  ...
```

Figure 4: TestGen Specification for Scenario Climb.

toResolver developers, to add more scenarios to exercise additional behavior of the system.

4. TEST INPUT GENERATION

By instantiating with meaningful values the parameters exposed by our test wrapper, it is possible to generate realistic test cases for AutoResolver. In this section, we present two different tools that we have developed for this purpose. The two tools have very different philosophies and characteristics, as described in detail below.

4.1 Black-Box Testing

To perform black-box testing, we developed a tool called TestGen. TestGen generates test cases in a black-box fashion, based on a declarative specification of the input space of the system under test. TestGen takes as input a method of some Java class and a description of ranges for primitive parameter values that are to be tested. It then generates test cases that correspond to the Cartesian product of the input value ranges.

Figure 4 illustrates a TestGen specification for the Climb scenario. Each scenario parameter is associated with a range together with a step for generating values to be tested within the range. For example, the heading values that are to be tested consist of 0.0, 10.0, 20.0, . . . , 180.0. In addition, TestGen also supports automatically filtering out uninteresting cases. For example, the very first line of the specification requires that the generated test cases include at least one aircraft with a non-zero climb rate. This is to avoid generating tests that are covered by the Cruise scenario.

TestGen automatically generates all possible combinations of the specified input values, filters out the uninteresting ones, and turns the remaining ones into JUnit tests that can be executed on the system under test. For the example of Figure 4, TestGen generated 24,016 tests.

In our previous work [16], we used the Java PathFinder[2] [36] open source model checking tool to generate black-box tests. In this work, we decided to avoid using a heavyweight model checking tool to solve this relatively simple problem. TestGen is intuitive to use. It can, if required, automatically parse Java code to identify public methods and their corresponding parameters, and create a *csv* spreadsheet for developers to enter their test specifications.

4.2 Concolic Testing

JDart [22, 17] is a concolic execution [20, 31] engine based on the Java PathFinder framework; it can be used to generate test cases exercising a large number of paths in a program. The tool executes Java programs with con-

[2]http://babelfish.arc.nasa.gov/trac/jpf

crete and symbolic values at the same time, and records symbolic constraints describing all the decisions (i.e., conditional branches taken) along a particular execution path. Combined, these path constraints form a *constraints tree*. The constraints tree is continually augmented by trying to exercise paths to unexplored branches. Concrete data values for exercising these paths are generated using a constraint solver. Even though we exclusively use Microsoft's Z3 SMT solver [10] at the moment, JDART is independent of a particular solver implementation thanks to a solver abstraction layer.

The efficacy of concolic testing stands—and falls—with the availability of a potent decision procedure for the respective theory in which the program constraints are formulated. As such, it seems unlikely that JDART would ever be able to cope with a huge, complex system like AUTORESOLVER. However, under the motto *failure is not an option*, we focused on improving the robustness of JDART to reap the benefits of improving coverage even under adverse conditions such as unsupported theories. The following paragraphs briefly describe the major challenges we faced, and how we were able to cope with or mitigate them.

Approximating Floating-Point Arithmetic. Unsurprisingly, AUTORESOLVER makes heavy use of floating-point calculations. At the time of publication, Z3 comes with only a very rudimentary (and undocumented) implementation of a floating-point theory. We quickly found that we would be unable to benefit from this, especially as common operations such as conversion between integers and floats are entirely unsupported.

Therefore, similarly to some previous work [24, 3], we chose a different approach of approximating floating-point values using reals. While this approximation is not sound, as it does not account for the limited-precision effects, it might frequently yield valuable solutions—even if they are *incorrect* (see below). Furthermore, in the latter case JDART can be instructed to "try harder" by imposing additional constraints. For instance, if a constraint involving an integer-to-float conversion yields an incorrect result, we will first try to restrict the solution space to those integers that can be losslessly stored in a floating-point number before giving up. Similarly, the limited-precision effects affecting arithmetic operations can be mitigated by restricting the relative difference of the operands involved. This, of course, increases the chance that the solver might return *don't know* or even *unsatisfiable* (which also needs to be interpreted as *don't know* because of the additional constraints we imposed), but this is no worse than the initial incorrect solution.

Another aspect concerns special floating-point values, such as NaN ("Not a Number") or infinity. These cannot be mapped to a value in the solver's theory on reals. In principle, we currently assume that these values do not occur. However, we intercept methods that are common sources of NaN or infinity results, such as `Math.asin`: if the argument is symbolic, we branch on whether it lies in the $[-1, 1]$ interval. As this branch is reflected in the constraints tree, we at least identify some paths on which NaN values occur, and can furthermore safely discard any symbolic information about the function result.

Handling Native Calls. Most complex JAVA programs make use of methods that need to be executed outside of the JVM, e.g., if they are provided by a library implemented in C. The Java PathFinder tool is not able to handle such *native calls* out of the box. Instead, native calls need to be reflected by so-called *native peers*, i.e., JAVA methods that are executed in the host JVM (the JVM executing Java PathFinder itself). Thanks to the recent `jpf-nhandler` extension,[3] these native peers can now be generated automatically on-the-fly for arbitrary third-party native methods.

However, we cannot keep track of symbolic information during the execution of native methods, as this does not happen under Java PathFinder's control. JDART can be instructed to deal with this in three different ways (which can be combined): (a) discarding all symbolic information and simply continuing with the concrete value returned, (b) symbolically annotating the return value with an uninterpreted function over method parameters,[4] or (c) performing symbolic execution through a *symbolic peer*, which may augment the constraints tree and symbolically annotate return values. These symbolic peers need to be implemented manually. For AUTORESOLVER, we used them to implement the above-described branching in `java.lang.Math` methods possibly returning NaN. For trigonometric functions and other mathematical functions we let JDART only annotate the return value symbolically with the function name. During constraint solving we make some simple assumptions about the ranges of these functions.

Dealing with Incorrect Solutions. The aforementioned problems frequently cause the solver to return solutions which, despite being correct modulo the employed theory, fail to exercise the intended concrete path in the program. Whether this is due to the unsound approximation of floating-point values by reals, or loss of symbolic information through a native call, when analyzing a program like AUTORESOLVER such incorrect solutions have to be considered the default rather than the exceptional case.

Simply discarding these solutions would hence be a huge waste of computation time. Instead, JDART analyzes every solution and tests if it can nonetheless be used to exercise *some* (even if not the originally intended) path that is still unexplored in the program. Hence, a computed concrete solution is discarded only when we are absolutely sure that no new information (i.e., paths) can be gained from it.

Selective Exploration. As described in Section 3.2, we employ a wrapper to generate complex loss of separation scenarios. This narrows down the input space from trajectories consisting of 300 three-dimensional coordinates each to a handful of floating-point variables, such as the airplane speed or heading. To symbolically keep track of the effects that these variables incur in the AUTORESOLVER code, the wrapper needs to be executed by JDART. JDART will then try to exhaustively explore the behavioral space of the wrapper combined with AUTORESOLVER.

However, given limited resources, it may be beneficial to focus on maximizing path coverage for AUTORESOLVER, and not necessarily the wrapper. To achieve that, JDART can be configured to explore the path-space *selectively* by suspending or resuming exploration upon method entry or exit. For example, we could configure JDART to not explore the initialization phase (i.e., when executing the scenario generation wrapper). Exploration could be specified to start when

[3] `https://bitbucket.org/nastaran/jpf-nhandler`

[4] For non-pure native methods this is incomplete, but there is no means of checking purity.

a method of any class in the main AutoResolver package `nasa.arc.aac` is invoked, and suspended again during methods of the uninteresting `DataLogger` class contained in this package. This is complemented by the possibility of specifying sets of input values for replay; JDart will replay these cases and explore according to the above criteria.

5. COVERAGE ANALYSIS

In the previous sections, we described our infrastructure for generating test suites for testing AutoResolver. The generated tests must be evaluated both in terms of how well they exercise the system under test, and also how realistic they are with respect to the problem that is being solved. We have established application-dependent measures of coverage to complement standard structural coverage criteria for AutoResolver. In this section, we describe these criteria and the tool support that we have developed for evaluating and comparing test suites accordingly. Moreover, we discuss how we have developed support for evaluating the generated tests within visualization tools developed by the AutoResolver team.

5.1 Coverage Metrics

Code Coverage. Among the multitude of code coverage criteria that exist [1], *branch coverage* has been identified as a good measure for comparison of test suite quality [18]. Moreover, *path coverage* (at the bytecode level), provides the most exhaustive structural coverage for exercising all behaviors of a system.

Branch coverage corresponds to the percentage of branching bytecode instructions (conditional jumps, or `if` and `switch` statements on the language level) that have been executed with both possible outcomes (jump or no jump). Each branching instruction is considered in isolation. Path coverage, on the other hand, also takes into account the combination of the different branching instruction outcomes in single executions.

Note that, in our experiments, we do not expect to achieve 100% coverage. First of all, our current scenarios do not cover all the possible cases that AutoResolver is designed to handle, such as arrivals and departures. It would be extremely hard for us to isolate the part of the code that deals with our scenarios in order to establish a coverage target. On the other hand, 100% coverage is very hard to achieve in general, since it is a purely theoretical value. For example, a program may contain unreachable code, the detection of which is, in the general case, undecidable. Some branches may not be satisfiable; determining whether the condition of a branching instruction is satisfiable requires at the very least a complete decision procedure for the underlying theory.

For these reasons, we report our coverage results both in terms of absolute numbers (of paths or branches covered) and percentages.

Resolution Coverage. Branch and path coverage measure the quality of a test suite at a very low level. A measure more targeted to the system under test is to look at the system's high-level functionalities. As described in Section 2, AutoResolver attempts different predefined maneuvers to resolve loss of separation, and executes the optimal maneuver among the successful ones. Hence, we introduce the *resolution coverage* metric by considering (1) maneuver types that result in *successful resolution attempts*, (2) maneuver types that are selected as the *executed resolution*, and (3) how many *unique combinations* (sets of successful maneuver types in order of application) of successful resolution attempts are exercised while running a test suite.

Table 1: Maneuver Types.

ID	Description
1	Temp. Altitude (climb to intermediate alt.)
3	Step Alt. (climb→level off→descend to original alt.)
4	Step Alt. (descend→level off→climb to original alt.)
5	Direct To (take a shortcut to route point)
8	Temp. Speed (temporarily increase/decrease speed)
13	Path Stretch (take a detour to route point)
15	Extend Altitude (extend current alt. for some time)
26	Offset (temporarily move to a parallel path)

The maneuver types attempted by AutoResolver in our generated scenarios are shown in Table 1. This is only a subset of all implemented maneuver types. Maneuvers not shown in the table are outside the scope of our current scenarios. For instance, they are related to losses of separation during arrivals or departures, or to conflicts involving weather, which we currently do not generate. Selection of relevant maneuvers was performed for us by the AutoResolver developers.

5.2 Analysis Tools

JaCoCo. JaCoCo is a free Java library for measuring code coverage. We use this library to record branch coverage on AutoResolver for the generated test suites.

CovComp. We developed CovComp to compare branch coverage of different test suites. The tool uses the JaCoCo library to measure branch coverage for multiple test suites and then visualizes the differences. It displays branch coverage for two individual test suites as well as for their union at the level of classes or for the actual source code. Lines with branching instructions in the code are colored according to their coverage by the two test suites and their union. We used CovComp for analyzing the differences in coverage between generated test-suites in detail.

AacViz. AacViz is a tool for visualizing and replaying resolutions produced by AutoResolver. It is used for debugging and analyzing the behavior of AutoResolver by its developers. To connect to AacViz, our wrapper and stub code implementations output logging information to an SQL database, which is the interchange format used by the AutoResolver development environment. We were thus able to use AacViz to visualize the trajectories that we generate in lieu of ACES, as well as trajectories that we generate in response to attempted maneuvers by AutoResolver (e.g., see Figure 3). Hence, with the help of the AutoResolver team, we were able to validate that our framework provides a useful and realistic enough, low fidelity, testing environment for the AutoResolver system.

6. EVALUATION

The components we have developed for testing AutoResolver are assembled into a three-phase testing workflow illustrated in Figure 5: data is shown as trapezoids and tools as rectangles.

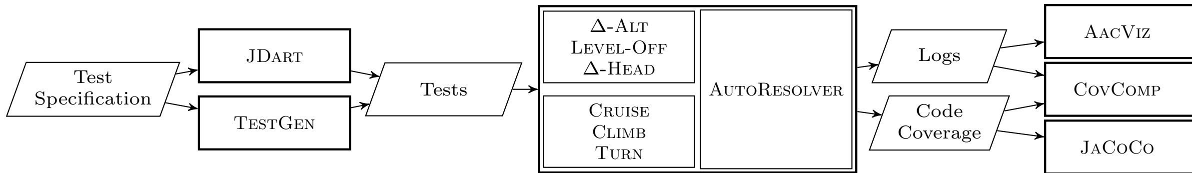

Figure 5: Workflow for Testing AutoResolver.

Table 2: Statistics and Path Coverage for JDart Scenarios.

Explore Wrapper	Scenario	No. of Params.	No. of Seeds	Depth Limit	Decisions per Path Shortest	Decisions per Path Longest	Path Coverage SAT	Path Coverage UNSAT	Path Coverage D/K	Runtime [Sec.]
No	Δ-Alt	1	2	∞	2,903	44,274	5	22,357	4	66
	Level-Off	1	4	6,500	6,252	76,268	4	491	0	411
	Δ-Head	1	3	5,000	78,723	149,844	3	843	1,527	2,178
Yes	Δ-Alt	*(as above)*		30,000	*(as above)*		5	33,132	19	78
	Level-Off			2,000			5	6,629	2,449	624
	Δ-Head			4,000			3	3,396	3,503	3,017

The first phase of the testing process consists of test-case generation. TestGen and JDart (see Section 4) both take as input a declarative specification for the target method, also prescribing ranges or sets of valid inputs to this method. The generated test cases target the wrapper scenarios for AutoResolver. Test-case execution and information logging happens during the second phase. We record branch coverage using JaCoCo, path coverage using JDart, resolution coverage using the AutoResolver's built-in logging mechanism, as well as several performance statistics, e.g., runtimes, sizes of test suites or path constraints. All obtained results and statistics are analyzed and compared during the third phase. We analyze and compare branch coverage using the reports generated by JaCoCo and CovComp. Interesting or odd test cases can be studied in detail using AacViz.

Our resulting framework is modular and extensible in all of its components. It can easily accommodate new test-case generation tools. Test specifications give us flexibility in changing the focus of test suites within the domains of the currently implemented scenarios. Additional scenarios can be incorporated for arrivals/departures and for weather conflicts. The abstract trajectories presented in Section 3.2 provide a robust basis for extending scenarios or defining entirely new ones. Currently, we execute test cases with JaCoCo to record code coverage. Since we generate JUnit tests, we could also use other Java tools and record coverage for alternative code coverage metrics.

In the following sections, we describe how we applied, evaluated, and validated our framework on AutoResolver.

6.1 Testing the AutoResolver

As discussed, our framework implements test scenarios that tame the input space of AutoResolver by providing input parameters that enable the generation of realistic trajectories. However, the input space of the scenarios is still infinite since input values such as heading and speed are continuous. Our test-case generation tools TestGen and JDart have fundamentally different philosophies for addressing this problem.

TestGen reduces the input space by enforcing discretization of the specified input ranges, through the specification of a step between values that it explores (see Figure 4). JDart, on the other hand, relies on path constraints to create equivalence classes of input values, where a single test case is generated as a representative of each class. Bounding the depth of exploration ensures that a finite number of equivalence classes is created. We expect that JDart will be successful in identifying tests that are not covered by TestGen, when the step defined for the discretization of ranges skips over values that are essential for covering specific paths in the program.

Generating Tests with TestGen. To apply TestGen, we define three scenarios that correspond to the ones discussed in Section 3.2, and specify value ranges and filters as follows. In the Cruise scenario, headings for one airplane range between 0 and 350 degrees with a step of 10 degrees. We use airspeeds from 400 to 600 knots at increments of 50 knots. The time to loss of separation is set to whole minutes with a minimum of 1 minute and a maximum of 10 minutes. The time offset for the arrival at the point of loss of separation is set from 0 to 40 seconds at intervals of 10 seconds, allowing for situations where no loss of separation occurs. Ranges for all parameters were selected with help of the developers of AutoResolver to create realistic concrete scenarios. A total of 9,000 test cases is generated as a result (cf. Table 3).

For the Climb scenario, we let one or both airplanes climb or descend at a climb rate of 1,500 feet per minute. Headings range between 0 and 180 degrees at intervals of 10 degrees. We do not use a time offset at the point of loss of separation, and we use the same range for the time to loss of separation as the Cruise scenario. Finally, we allow both airplanes to start their climb/descent at multiples of 120 seconds and end the climb/descent at multiples of 40 seconds. For Climb, we include a filter that ensures that airplanes always cruise at multiples of 1,000 feet (commercial aircrafts typically do this) and do not climb above 35,000 feet. Moreover, to avoid recreating Cruise scenarios, we use a filter to ensure that at least one airplane climbs or descends. Altogether, this results in 35,568 test cases, of which 34,784 exhibit loss of separation (cf. Table 3).

For the Turn scenario, we use the same time and initial heading ranges as for the Climb scenario. We allow both

Table 3: Resolution Coverage of Maneuvers by Test Suites per Scenario.

Tool	Scenario	No. of Params.	No. of Tests	No. of Conflicts	Successful Resolutions	Executed Resolution	Unique Comb.
TestGen	Cruise	4	9,000	5,881	3, 4, 8, 13, 26	3(90.4%), 26(7.5%), 13(2.1%)	12
	Climb	8	35,568	34,784	1, 3, 4, 13, 15, 26	15(72.2%), 3(19.6%), 4(4.8%), 13(3.1%), 26(0.3%)	34
	Turn	6	72,960	49,484	3, 4, 5, 13, 26	5(47.3%), 3(35.7%), 13(16.4%), 26(0.6%)	21
JDart	Δ-Alt	1	5	1	3, 4, 13, 26	3(100.0%)	1
	Level-Off	1	4	3	4, 13, 15, 26	26(100.0%)	1
	Δ-Head	1	3	3	3, 4, 5, 13, 26	5(66.7%), 3(33.3%)	2

airplanes to change heading between −60 and 60 degrees at a step of 30 degrees. Changes in heading are allowed at multiples of 2 minutes. Our filters require that at least one of the airplanes changes heading, thus avoiding to recreate test cases covered by the Cruise scenario. With 72,960 test cases, this is the largest test suite we generate. As discussed, the filters that we use also ensure that the test suites for the three scenarios are mutually disjoint (cf. Table 3).

Generating Tests with JDart. To apply JDart, we similarly define three scenarios; these are small variations of the scenarios for TestGen, or limited versions of them necessitated by the scalability issues of this more sophisticated tool.

Δ-Alt fixes two airplanes in level flight. The only parameter exposed to JDart is the altitude difference at the time of loss of separation. This is a variation of Cruise, where the altitude difference was fixed to zero. We also constrain airplane altitudes to multiples of 1,000 feet, similarly to TestGen.

Level-Off is a variation of the Climb scenario. We fix two airplanes in a situation where one airplane is in level flight and the other is climbing for some stretch of time. We let JDart control the altitude difference at the time of loss of separation. The effect in this case is different than in the Δ-Alt scenario: since one airplane is in the process of climbing at the time of loss of separation, this airplane does not have to be at a multiple of 1,000 feet when the loss of separation occurs. We only constrain the inputs that JDart generates to an interval from −2,000 feet to 2,000 feet.

Δ-Head is a variation of the Turn scenario. We fix all parameters except for the change of heading for one of the airplanes, i.e., JDart controls the new direction of the airplane's flight. Similarly to the Turn scenario, the change of heading ranges between −60 and 60 degrees.

We use two different configurations in our experiments with JDart. In the first, we limit the initial exploration until the execution enters AutoResolver; in the second, we do not limit the start of the exploration, i.e., we explore both the wrapper and AutoResolver, but, as a result, impose stronger limits on the depth of exploration (i.e., the depth up to which we negate path constraints to find new executions). One configuration explores more paths, analyzing shorter prefixes of path constraints, while the other one explores fewer paths but analyzes path constraints to a greater depth. In both configurations we use a small number of manually selected input values (seeds) as starting points for the concolic execution.

Results. Table 2 reports statistics for test-case generation with JDart. Note that we include both configurations, i.e., with and without wrapper exploration. The table shows the number of parameters for each scenario, the number of seeded input values, the manually fixed depth bound, the number of satisfiable paths (used to generate test cases), the number of unsatisfiable path prefixes, the number of inconclusive (*don't know*, D/K) path prefixes, and runtimes. Additionally, we ran JDart without a constraint solver and only recorded the length of path constraints for the found test cases. We report the length, in terms of number of conjuncts, of the shortest and longest respective path constraint.

For all three scenarios, exploration proceeds faster and deeper into AutoResolver when restricting exploration of the wrapper. However, the test cases found in this deeper exploration are subsumed by the ones found in shallow exploration. In general, few test cases are generated; as discussed in Section 4.2, the high number of unsatisfiable path prefixes is due to the exclusion of special floating-point values NaN and infinity (methods `isNaN()` and `isInfinite()` are used heavily in great circle computations). High numbers of inconclusive path prefixes are seen for the Δ-Head scenario, where symbolic parameters are often used in trigonometric computations.

Table 3 shows the test suites that were generated for all six scenarios and how the generated tests cover the maneuvers that AutoResolver uses to resolve conflicts. For each scenario, we report the number of parameters exposed to the test case generation tool (TestGen for the upper part of the table and JDart for the lower part), the number of tests generated, and the number of test cases exhibiting loss of separation. We then report the set of successful resolutions for each test suite, i.e., the classes of maneuvers that resolve the loss of separation. (Note that AutoResolver typically generates multiple successful resolutions for each test case.) Among these resolutions, AutoResolver selects a single one to be executed (executed resolutions). The table reports the types of executed resolutions and the percentage of test cases in the suite where they occur. Finally, we report the number of unique combinations of successful resolutions (see Section 5).

Unsurprisingly, TestGen generates much larger test suites than JDart since it is cheap, fast, and unsophisticated—it does not explicitly target coverage and does not attempt to generate minimal test suites. Each TestGen test suite (upper part of Table 3) includes unique successful resolutions. *Temporary Speed* (8) maneuvers are only successful in Cruise test cases. This maneuver became possible for this scenario when we exposed as a parameter the offset in time with which the airplanes arrive at the point of loss of separation. Larger offsets, in combination with a temporary change in airspeed, lead to both airplanes passing this point with enough space to ensure separation. *Temporary Altitude* (1) and *Extend Altitude* (15) maneuvers are only successful in Climb scenarios since they do not apply to airplanes in cruise flight. Finally, *Direct To* (5) maneuvers are only successful in Turn, where the angle formed in the trajectory by

Table 4: Test Execution Times and Branch Coverage for a Total of 10,568 Branches.

Scenario	No. of Tests	Execution [Sec.]	Branch Coverage [No.]	[%]
CRUISE	9,000	333	1,800	17.0
CLIMB	24,016	1,101	2,000	18.9
TURN	72,960	3,451	1,843	17.4
Δ-ALT	5	1	1,668	15.8
LEVEL-OFF	5	2	1,628	15.4
Δ-HEAD	3	2	1,663	15.7

the change of heading can be shortcut by omitting a point in the future route. The test cases generated with JDART for the restricted scenarios cover only a subset of the successful resolutions, executed resolutions, and unique combinations covered by the corresponding TESTGEN generated tests for the more general scenarios.

Finally, Table 4 shows execution times and achieved branch coverage for all six test suites. As can be seen, in contrast to the unique combinations coverage, the difference in percentual coverage between the smallest test suites (with only three test cases) and the larger ones is modest. This indicates that branch coverage may not be a good metric of behavioral coverage for this application: AUTORESOLVER tries a limited set of maneuvers with different parameters in changing order. All test cases exercise almost the same set of maneuvers (maneuvers are tried to learn that they are not successful). Additional coverage comes from (few) branches when choosing parameters and rating successful maneuvers.

6.2 Discussion

The testing framework that we developed for AUTORESOLVER relies on four hypotheses that we assess in this section based on our experimental results.

H1. *The* ACES *stubs (a) allow for a light-weight and targeted approach to testing* AUTORESOLVER*, and (b) are precise enough to yield meaningful results (see Section 3.1).*

Evaluation. The hypothesis has two parts. Our experiments show that we can run thousands of tests with a high percentage of conflicts in minutes. Our environment accepts small, approximate test cases, as opposed to fully detailed scenarios with thousands of flights (not all of which are conflicts) corresponding to airport data.

The second part, namely that our stubs for ACES generate trajectories with sufficient precision, is harder to assess without the help of domain experts. We tested this hypothesis by logging trajectory and resolution data from test cases and analyzing this data together with the developers of AUTORESOLVER (five trajectories for each of the three scenarios). By expert judgment, our implementation of maneuvers and generation of trajectories is good enough for testing AUTORESOLVER without ACES.

H2. *Scenarios are sufficiently complementary and generic to allow for exercising a relevant subset of the behavior in* AUTORESOLVER *(see Section 3.2).*

Evaluation. To evaluate this hypothesis, we compared the coverage contributed by each scenario. Figure 6 visualizes the differences and overlap in coverage. It confirms the hypothesis by showing that no scenario completely subsumes another scenario in terms of coverage. Chart (a) compares the overlap in unique combinations of successful resolutions for the test suites generated with TESTGEN. For every pair of scenarios, the lined and dotted parts of the bars represent unique combinations that occur only in one scenario, while the white area represents the number of unique combinations that are exercised by both scenarios.

As can be seen, the overlap between the scenarios is significantly smaller than the number of individually covered combinations. This indicates that each scenario contributes to the overall coverage of AUTORESOLVER. Chart (b) visualizes the overlap in branch coverage in the same fashion. In this case, the overlap between scenarios is significant. As mentioned, this indicates that branch coverage may not be a good metric of behavioral coverage for this application.

H3. *The parameterization of scenarios with only a few primitive parameters is sufficient for generating test cases of high quality (see Section 3.2).*

Evaluation. At the current stage, we measure quality mainly in terms of coverage, mostly concentrating on behavioral coverage at the level of resolutions since this is an important criterion for the AUTORESOLVER team. In the future, we plan on extending our work towards evaluating error-finding capabilities of generated test cases.

The parameters that we expose control the airplanes in the scenarios and enable us to create test cases that emulate realistic situations. We checked our results for consistency against the expectations of the developers of AUTORESOLVER. They confirmed, for example, that *Step Altitude* maneuvers are expected to be chosen at a high rate (90%) in the CRUISE scenario as these maneuvers create very little delay.

Further evidence that the hypothesis holds is provided by the recorded branch coverage. We analyzed the coverage for methods in the central classes of AUTORESOLVER: though branch coverage overall is low, for the targeted methods (corresponding to maneuvers), the coverage is much higher (approximately 50% on average). Uncovered branches are mostly related to null checks on method parameters, and to parameters of airplanes that we do not currently control in our scenarios (e.g., flags that describe if an airplane can maneuver). These flags could easily be exposed in additional scenarios.

H4. *The implementation of parameterized scenarios as an interface between the tester and* AUTORESOLVER *tames the input space sufficiently to make* AUTORESOLVER *amenable to automated test case generation techniques (see Section 3.2).*

Evaluation. This hypothesis is confirmed through the mere fact that we were able to generate realistic input trajectories for AUTORESOLVER with both black- and white-box techniques. Moreover, the other three established hypotheses speak to the fact that the generated test cases are not only realistic, but also useful for exploring the behavioral space of this complicated system.

While we have spent considerable effort on making JDART robust enough to run on AUTORESOLVER, i.e., record and analyze single paths, only the restrictions defined through the scenarios drastically reduced the number of paths to be analyzed.

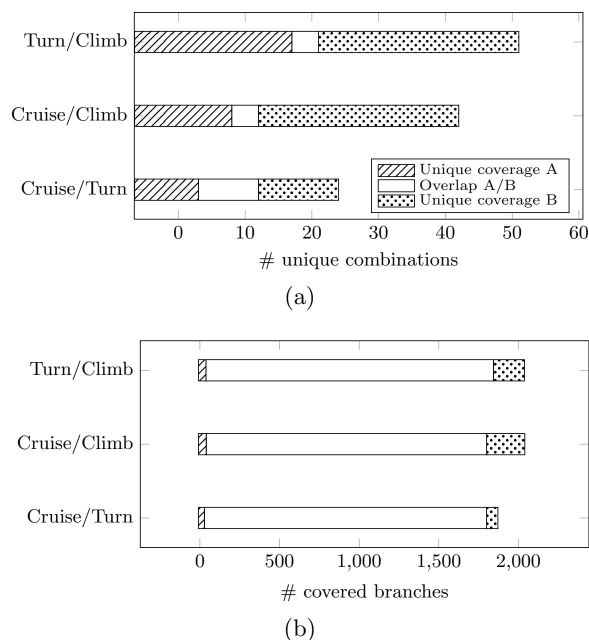

Figure 6: Coverage Comparison.

7. RELATED WORK

AutoResolver has previously been integrated and evaluated with other National Airspace System (NAS) simulations which have non-zero trajectory prediction errors [26, 29, 33]. In contrast to these simulation-based approaches, we address the problem of test-case generation.

The generation of structurally complex inputs is targeted by several researchers. In a black-box setting, frameworks like Korat [6] and Udita [19] support test case generation through declarative specifications of the test inputs. In a white-box setting, SAGE [21] is a fuzzer for security testing based on concolic execution, applied successfully to Microsoft systems such as media players and image processors. In [2], ideas from procedural content generation, an automated approach to generating content for computer games, are used to generate complex test scenarios.

The generation of method sequences for object-oriented systems is another focus of current testing research. In this context, black- and white-box techniques are combined for the generation of sequences and data values for primitive method parameters, respectively [35, 23, 34]. JPF-Doop [11] combines the Randoop approach [27] of feedback-directed random testing with concolic execution (JDart) to improve code coverage of testing Java software components. Garg et al. [14] take a similar approach but for C/C++.

MACE [8] combines black- and white-box techniques (active automata learning and concolic execution) to build an abstract model of an application under test in order to increase code coverage and exploration depth. MACE targets testing of the protocol between a system under test and its environment, and as such is not directly applicable to AutoResolver. However, our idea of restricting the initial exploration of JDart in order to reach deeper into the AutoResolver code is similar in spirit to the MACE's approach.

Existing test-case generation approaches can be added to our testing framework for AutoResolver, e.g., plain random input generation, or evolutionary test case generation [28], which targets large input domains and has been used successfully in industrial applications [7]. However, as already explained, it is impossible for any such tools to directly tackle the AutoResolver input space. Therefore, they would have to be used as prescribed by this paper, through our wrapper that tames the input space of the problem. Note that the fact that AutoResolver makes heavy use of nonlinear arithmetic and floating-point operations presents a major challenge for many existing tools.

Finally, some works focus on guiding or restricting test case generation by program invariants inferred from executions [5, 9]. While our filters and input ranges are currently derived manually from expert knowledge, it would be interesting to investigate the potential of inferring these from simulations of AutoResolver on recorded flight data.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we described collaborative work of several years between formal methods and domain experts in generating a light-weight testing environment for a complex system for separation assurance called AutoResolver. The main challenge of this project has been to find a way to tame the input space of AutoResolver. We achieved this through the implementation of parameterized scenarios that make the input space amenable to meaningful test case generation.

We developed a modular, extensible framework that puts together several tools and techniques for test case generation, execution, and evaluation. Specifically, we stubbed out ACES, developed TestGen and CovComp, put major effort into robustifying and adding features to JDart, and implemented logging to connect to AacViz used by the AutoResolver developers. Our efforts have paid off: we have been able to generate thousands of meaningful test cases that run in a matter of minutes.

In the future, we plan to extend the currently supported test scenarios, introduce secondary aircrafts, as well as experiment with new ways of combining our test-case generation techniques. For example, our experiments show that, when limiting the exploration of the wrapper by JDart, the space of behaviors that can be explored can be very narrow unless the JDart seed values are chosen carefully. It would be interesting to explore an approach where the tests generated by TestGen are used to create different paths for entering the AutoResolver code for subsequent concolic exploration.

Finally, our experiments have shown that trigonometric constraints still limit the scalability of our concolic approach, even though we have made advances in handling them. For example, for the very restricted Δ-Head scenario, where only the heading change of one airplane is explored, there are 1,527 constraints that could not be solved, as opposed to 4 and 0 for the other scenarios under the same configuration (see Table 2). We plan to investigate additional techniques for dealing with trigonometric constraints such as using randomized constraint solvers [32].

Acknowledgements. We thank CMU SV students Mariam Rajabi and Norman Xin for assisting with the development of the CovComp and TestGen tools. We also thank Heinz Erzberger for initiating this collaboration.

9. REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

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

[3] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 549–560, 2013.

[4] D. Bell, F. Kuehnel, C. Maxwell, R. Kim, K. Kasraie, T. Gaskins, P. Hogan, and J. Coughlan. NASA World Wind: Opensource GIS for mission operations. In *IEEE Aerospace Conference*, pages 1–9, 2007.

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

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

[7] O. Bühler and J. Wegener. Evolutionary functional testing. *Comput. Oper. Res.*, 35(10):3144–3160, Oct. 2008.

[8] 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 *USENIX Security Symposium*, 2011.

[9] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology*, 17(2):8:1–8:37, 2008.

[10] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[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 *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 *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 *International Conference on Software Engineering (ICSE)*, pages 132–141, 2013.

[15] S. George, G. Satapathy, V. Manikonda, K. Palopo, L. Meyn, T. A. Lauderdale, M. Downs, M. Refai, and R. Dupee. Build 8 of the airspace concept evaluation system. In *AIAA Modeling and Simulation Technologies Conference*, 2011.

[16] 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.

[17] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. In *International Static Analysis Symposium (SAS)*, pages 248–264, 2012.

[18] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 302–313, 2013.

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

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

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

[22] F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 268–279, 2013.

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

[24] F. Ivancic, M. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 49–58, 2010.

[25] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

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

[28] 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.

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

[30] C. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and

analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 11:339–353, 2009.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[32] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. CORAL: Solving complex constraints for symbolic PathFinder. In *NASA Formal Methods Symposium (NFM)*, pages 359–374, 2011.

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

[34] 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.

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

[36] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.