Formal Testing for Separation Assurance

Dimitra Giannakopoulou[1], David H. Bushnell[2], and Johann Schumann[3]

[1] CMU / NASA Ames
[2] TRACLabs / NASA Ames
[3] SGT / NASA Ames

Abstract. In order to address the rapidly increasing load of air traffic operations, innovative algorithms and software systems must be developed for the next generation air traffic control. Extensive verification of such novel algorithms is key for their adoption by industry. Separation assurance algorithms aim at predicting if two aircraft will get closer to each other than a minimum safe distance; if loss of separation is predicted, they also propose a change of course for the aircraft to resolve this potential conflict. In this paper, we report on our work towards developing an advanced testing framework for separation assurance. Our framework supports automated test case generation and testing, and defines test oracles that capture algorithm requirements. We discuss three different approaches to test-case generation, their application to a separation assurance prototype, and their respective strengths and weaknesses. We also present an approach for statistical analysis of the large numbers of test results obtained from our framework.

1 Introduction

The Federal Aviation Administration (FAA) and industry forecast that air traffic operations are expected to increase 150 to 250 percent over the next two decades [1]. The Next Generation Air Transportation System (NextGen) is a NASA research program contributing to a larger NextGen program of the FAA and the Joint Planning and Development Office (JPDO). NextGen addresses the increasing load on the air traffic control system through innovative algorithms and software systems. The seminal work produced by this program should ideally be in the form of reference artifacts that can be adopted by industry.

It would be desirable for the algorithms developed to be in some abstract, semantically clear, and implementation independent form. Abstract specifications concentrate on the key aspects of algorithms, so it is easier to argue about their correctness. They also allow freedom of implementation, which is important since algorithms may be deployed in different contexts. However, in our experience, reference artifacts often take the form of executable programs that implement the intended functionality of the algorithms. These programs are written in languages such as Java, C/C++, or Python, that unavoidably include lower level details that may interfere with the analysis. On the other hand, developers are more familiar with them and it is easier to capture a fine level of detail in such languages.

This paper reports on collaborative work between the Robust Software Engineering (RSE) group and the NextGen group at the NASA Ames Research Center. This work aims at the development of techniques and processes to ensure creation of robust software prototypes for separation assurance algorithms. More specifically, we have focused our initial efforts on testing for the Tactical Separation Assisted Flight Environment (TSAFE) component of NextGen. TSAFE is the part of NextGen which seeks to predict and resolve loss of separation between aircraft in the 30 second to 3 minute time horizon. Although our efforts currently target separation assurance, our long term goal is to demonstrate that our techniques are applicable to several classes of air traffic control software.

We have developed a testing framework for automatically generating and executing test suites for separation assurance, and have identified test oracles that capture correctness requirements against which the algorithms are tested. Developing such a testing framework for separation assurance algorithms is a challenging task. One of the characteristics of air traffic applications is that they involve complex inputs. For example, in the separation assurance prototype that we study [2], inputs include position, velocity, and direction of multiple aircraft, as well as whether they are eligible for change in course as directed by air traffic controllers. It is extremely hard to manually create enough realistic inputs to thoroughly exercise these applications. For this reason, developers and researchers typically use airport data recordings as test inputs. Their correctness and safety claims are based on demonstrating that the software does not exhibit errors while running with real inputs for a certain amount of time. However, as a loss of separation between aircraft occurs rarely, most airport data recordings only represent nominal scenarios that do not cover enough parts of the code that handles off-nominal cases.

On the other hand, confidence in the system depends on how it reacts to such unexpected scenarios, leading to the question of how much testing is required to achieve this confidence. In practice, and depending on the System under Test (SuT), a system is tested until some degree of test coverage is established. The coverage criterion depends on the V&V process that needs to be followed, which may involve compliance to standards defining minimum coverage requirements. For example, flight software is tested to achieve some predetermined degree of Modified Condition/Decision Coverage (MCDC) [3]. Other examples of relevant metrics are the more simple statement coverage, or the potentially expensive path coverage. The more stringent a coverage criterion, the more test cases are typically needed to satisfy it. The required number of test cases to achieve certain coverage can easily exceed what can be created manually.

Another challenging task in the verification of separation assurance algorithms is the identification of test oracles. For example, it is very difficult to characterize what is expected of an algorithm that proposes a change of course for aircraft that may lose separation. The purpose of such algorithms is not simply to select some course that will ensure avoidance of loss of separation, but

to select the best possible solution based on a set of criteria. Formulating test oracles for such optimization problems is a non-trivial task.

This paper reports on our approach to addressing the afore-mentioned challenges for a component of TSAFE[1] that is responsible for computing solutions when loss of separation is detected. More specifically, the paper is structured as follows. Some background on the verification of SA algorithms is provided in (Section 2). We then present three different approaches to generating test cases automatically. The first approach is based on symbolic execution of the prototype code, and is discussed in Section 3. The second approach, presented in Section 4 is based on the use of a model checker for generating tests for the application as a black box. The third approach, described in Section 5 is a combinatorial testing approach that tries to cut down on the combinations of input values, for scalability. Our approach to formulating requirements as well as the requirements that we identified for TSAFE are presented in Section 6. The practical framework that we have set up for testing TSAFE is discussed in Section 7, followed by the results obtained from our testing, and the relative strengths and weaknesses of our proposed approaches in Section 8. An alternative approach for examining the results of the large amount of date generated from our testing process is presented in Section 9. Finally, Section 10 closes the paper with conclusions and plans for future work.

2 Verification of Separation Assurance

Advanced verification and validation (V&V) techniques play a central role in ensuring important characteristics for the reference artifacts that will be delivered to the NextGen program. Our approach to verification and validation aims at the development of V&V processes and techniques to:

- Ensure creation of robust software prototypes for key algorithms.
- Provide quantifiable criteria for automated conformance verification of production code with these prototypes.

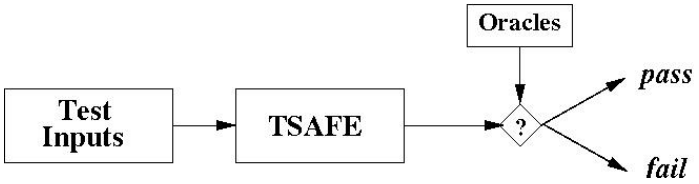

Fig. 1. Targeted testing framework for TSAFE

[1] For conciseness, we will often refer to our targeted component as TSAFE in the rest of the paper.

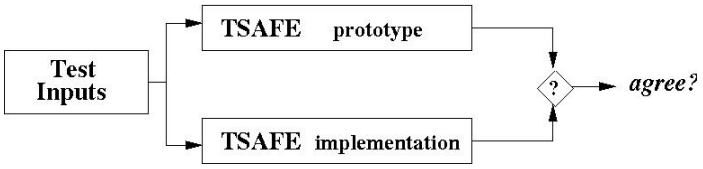

Fig. 2. Using prototypes for conformance testing

To achieve these goals, we aim at developing test suites and test oracles against which the prototypes will be checked for correctness (see Figure 1). Test suites are sets of test cases (i.e., test inputs) aimed at exercising the behavior of the system under test. A quality test suite also provides a good criterion for conformance testing, since it can be used to check whether production code and reference prototypes have matching results on the tests included in the suite (see Figure 2). Test oracles are the "properties" or "requirements" set by the algorithms and expected from the prototypes.

Algorithm and input space. The basic separation assurance problem involves two aircraft that may lose separation within a few minutes of flight. The problem of course becomes significantly more complicated when multiple aircraft are involved. The approach taken in the latter case is to still target the loss of separation problem for two airplanes (called "primary"), but to solve the problem in the presence of secondary aircraft. The way the algorithms work in this case is to select, among a number of potential solutions, not the one that is necessarily best for the primary aircraft, but one that also avoids, as much as possible, creating loss of separation with secondary aircraft.

Our approach to verification of separation assurance has been to start by thoroughly analyzing the basic problem involving only the two primary aircraft, and then gradually introducing secondary aircraft, one at a time. Some of the requirements may of course change for the basic and extended version of the problem. However, since the extended version depends on correct operation of the basic version, it makes sense to spend a significant amount of the effort on analyzing the basic case.

The particular code that we have been analyzing also tries to resolve horizontal loss of separation on a two dimensional space. It is customary for horizontal and vertical loss of separation to be resolved separately. Our targeted code therefore involves the following inputs for each aircraft (primary or secondary): its X and Y coordinate values, velocity and its direction of flight. Moreover, for the primary aircraft, because air-traffic controllers may or may not be allowed to advise a change of course, a boolean variable is also used that reflects turn eligibility.

Test input generation. Testing may be carried out in a systematic or random fashion, the latter one using randomly generated test inputs to exercise the code. Research shows that random test input generation can have good coverage. However, a random approach may not be able to drive a program through spe-

cific paths, and tends to generate many redundant test cases. In the context of NextGen applications such as TSAFE, random testing may also generate inputs that are not meaningful. For example, it could produce a trajectory that cannot be flown by a real airplane. Systematic testing, on the other hand, tries to generate an efficient set of test cases that will achieve the desired testing goal. As discussed in Section 1, the size and expected quality of test suites easily exceeds what humans can deliver without the support from automated tools. Our approach therefore consists of generating test cases in a systematic and automated fashion.

In particular, we investigate and apply three different approaches for this purpose. The first approach, based on symbolic execution of the prototype code, generates ideal test suites, when successful. By ideal we mean that it avoids redundancy while ensuring the coverage criterion that is required of the test suite. The second approach is based on the use of a model checker to systematically generate the interesting combinations of inputs for a particular problem, where interesting in our case is defined by the developers of the algorithms. Clearly, as more secondary aircraft are introduced into the problem, the combinations of inputs becomes too large to explore exhaustively. We therefore also investigate a third approach, which tries to limit the size of a test suite through combinatorial testing, a technique that generates only some combinations on the input space, while trying to maintain the quality of a test suite.

In the following sections, the three approaches are discussed in detail. A main component in some of these approaches is the JavaPathfinder model checker, which we briefly describe below.

JavaPathfinder (JPF). JPF [4, 5] is an open source verification framework developed by the RSE group at NASA Ames. It has been started as an explicit state model checker for Java bytecode. A model checker explores all the possible states that a program may be in and the transitions between these states, in a systematic and intelligent manner [6]. The focus of JPF is on finding bugs, such as concurrency related bugs (deadlocks, races, missed signals, etc.), runtime related bugs (e.g., unhandled exceptions), and others. JPF can also check for violations of user-specified assertions that encode application specific requirements. JPF uses a variety of scalability enhancing mechanisms, such as user-extensible state abstraction and matching, on-the-fly partial order reduction, configurable search strategies, and user definable heuristics (searches, choice generators).

Related Work. The NextGen program relies heavily on innovative algorithms and software systems that will require rigorous certification and assurance. Several efforts have been performed towards the specification and verification of such algorithms and systems.

Leveson et al. [7] present an approach to writing requirements specifications for process-control systems,and apply this approach to an industrial aircraft collision avoidance system (TCAS II). Their focus is on using a specification language that is readable and reviewable by applications experts who are not computer scientists or mathematicians. Writing specifications of a system in easily readable, yet formal, languages is an important step towards achieving rigorous implemen-

tations. Detailed specifications may be analyzed by automated tools and could potentially be used as the references that guide the actual implementations by contractors. They capture the key functionality of a system, and are independent of implementation platform / language. However, the state of the practice is that researchers tend to develop their prototypes in programming languages. Writing specifications require abstraction skills and familiarity with languages that developers may not have the time to acquire. Betin-can and Bultan [8] present a design for verification approach where they propose patterns of synchronization to be used in concurrent programs in order to facilitate verification through model checking. They have applied their approach in the analysis of a concurrent implementation of TSAFE within a test-bed for experimenting with approaches to achieving high dependability systems. The implementation of TSAFE that we are working with is not concurrent. Moreover, we focus our efforts on generating tests, rather than verifying TSAFE exhaustively. As discussed earlier, tests may be used both to establish robustness of a reference prototype, and as a way of establishing conformance between software system.

There exists work on mathematically proving correctness of key algorithms for air traffic control [9, 10]. Although theorem provers can semi-automate this process, proving correctness in this way can only be done for select safety critical parts of a research prototype. The majority of the verification will still be carried out through extensive testing. Automatic test case generation is crucial to reducing the cost and increasing the efficiency of testing. Finally, some of our early experimentation with symbolic execution in this domain for a different TSAFE prototype is presented in [1].

3 White-Box Test-Case Generation with Symbolic Execution

In this section, we discuss test-case generation techniques based on symbolic execution. These are sophisticated and powerful techniques that theoretically have the potential to automatically generate test inputs for reaching specific locations in the prototype code. As such, they can be used to build quality test suites for achieving desired types of structural / behavioral coverage of the code. We briefly describe the basic idea behind these techniques, as well as their practical limitations.

Symbolic Execution. Symbolic execution [11] is a program analysis technique that uses symbolic values instead of actual data as inputs to the program to be executed; symbolic expressions represent the values of program variables. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. The state of a symbolically executed program includes the (symbolic) values of program variables, a path condition (PC), and a program counter. The path condition is a boolean formula over the symbolic inputs, encoding the constraints that the inputs must satisfy in order for an execution to follow the associated path. Path conditions can be solved using off-the-shelf constraint solvers, to generate test cases (pairs of test input and expected output)

that are guaranteed to exercise the analyzed code. The paths followed during the symbolic execution of a program are characterized by a symbolic execution tree. To illustrate the difference between concrete and symbolic execution, consider the following example:

```
if ((xB0 > 0 && psiB > 0 && psiB < Math.PI) ||
    (xB0 < 0 && psiB > Math.PI && psiB < 2.0 * Math.PI)) {
  ... /* handle aircraft diverging */
} else {
  ... /* handle aircraft converging or parallel */
}
```

This code checks whether the aircraft are diverging (headed away from each other so there is no chance of a loss of separation) or converging (headed towards each other so they might have a loss of separation).

In the TSAFE simulations, one aircraft is always placed at the origin with a heading of 0 radians. In the above code example, `xB0` is the second aircraft's initial x coordinate and `psiB` is its heading in radians. These two variables are the inputs for this expression.

In concrete execution (i.e., in normal testing), code is executed on given concrete inputs. For example, when `xB0` = 20.0 and `psiB` = $\pi/3$, only one path through the code will be executed, corresponding to the first disjunct in the if-statement being true. In contrast, symbolic execution starts with symbolic input values `xB0` = $SymX$, and `psiB` = $SymPsi$. Symbolic execution will discover eleven paths through the program, corresponding to the eleven different ways the Boolean subformulas in the condition of the if-statement can be true or false and will generate eleven path conditions, according to different possibilities in the code:

```
psiB < 3.141592653589793 && xB0 < 0.0

3.141592653589793 == psiB && xB0 < 0.0
...
psiB > 3.141592653589793 && psiB > 0.0 && xB0 > 0.0
```

Note that symbolic execution is applied at the bytecode level, where all complex logical expressions get decomposed into simple ones. This is why it will generate path conditions for every combination of valuations of the expression's basic components. Concrete values for the inputs that satisfy ("solve") the path conditions are then found with the help of a constraint solver. Table 1 show the eleven test cases for this example. These solutions are subsequently used as concrete test inputs that are guaranteed to give full path coverage for this code. Note that the coverage that can thus be achieved by symbolic execution implies all other notions of structural coverage, including branch or MC/DC [3] coverage. In the following, we describe the automated tools that the RSE has developed for symbolic execution, and their application to TSAFE.

Number	psiB	xB0
1	1.5207958267948964	-10.0000005
2	3.141592653589793	-10.0000005
3	4.71238898038469	-10.0000005
4	6.283185307179586	-10.0000005
5	6.298893770447535	-10.0000005
6	*don't care*	0.0
7	-0.0500005	10.0000005
8	0.0	10.0000005
9	1.5707963267948966	10.0000005
10	3.141592653589793	10.0000005
11	4.728097443652638	10.0000005

Table 1. Test cases as generated by SPF from the above example.

3.1 Symbolic PathFinder (SPF)

SPF [12] implements a non-standard interpreter for bytecodes on top of the JPF model checker. The symbolic information is stored in attributes associated with the program data, which are propagated on demand, during symbolic execution. The analysis engine of JPF is used to systematically generate and explore the symbolic execution tree of the program. JPF is also used to systematically analyze thread interleavings and other forms of non-determinism that might be present in the code. Furthermore, JPF is used to check properties of the code during symbolic execution. Off-the-shelf constraint solvers and decision procedures such as choco [13] and IASolver [14] are used to solve mixed integer and real constraints. Loops are handled by putting a bound on the model-checker search depth and/or on the number of constraints in the path conditions.

Previous experience with SPF has shown that the application of automated test-case generation techniques may significantly improve the testing process [12]. For example, SPF was used to test a Java model of the Crew Exploration Vehicle's prototype ascent abort handling software, the On-board Abort Executive (OAE). Manual testing of the component took more than 20 hours and did not achieve the testing coverage desired by the developers. Random testing, on the other hand, achieved very poor coverage due to the large input state space and complex logic structure the code. In contrast, the symbolic execution framework generated approximately 200 test cases to obtain full coverage in less than one minute. The generated test cases helped to uncover subtle errors in the code, which were later corrected by the developer.

3.2 Application of SPF to TSAFE

The TSAFE prototype that we analyzed has been developed in Java, so SPF can be applied directly to it. On the other hand, the TSAFE code has many characteristics that are challenging for symbolic execution. Most calculations

use floating point arithmetic and include highly non-linear operators involving squaring, square roots, and trigonometric functions. Many of the calculations involved cannot be handled by constraint solvers. Moreover, the code contains nested loops. Finally, the TSAFE algorithm is implemented entirely in the class constructors, which are consequently relatively large.

We therefore decided to first experiment with applying SPF to a small part of TSAFE, namely the class *Conflict_probe*. *Conflict_probe* tests for conflicts between two aircraft within a fixed time horizon. The two aircraft can be both primary or one primary and one secondary. The aircraft can be flying level or turning. If both are turning, they are assumed to turn for the same length of time. *Conflict_probe* is self-contained, and uses no other classes from TSAFE. It involves trigonometric functions and nested loops. Therefore, even though *Conflict_probe* is a small part of the algorithm, it contains significant calculations, and can be viewed as a representative component of the TSAFE prototype.

In order to apply SPF to *Conflict_probe*, we needed to make the constructor's inputs symbolic and also limit the search depth for handling the loops. Moreover, we needed to manually modify some of the involved mathematical functions into equivalent code that could be handled by the constraint solvers. Despite these efforts, even for this restricted part of TSAFE, SPF generates path conditions that are too complex for the constraint solvers to solve. The principal problem was the length of the code: the path conditions generated by many hundreds of lines of code are inevitably complex. We did have some success, though, by further modularizing the code. Part of the *Conflict_probe* algorithm calculates the time when the two aircraft will reach minimum separation. After extracting this code into its own method we were able to successfully run SPF on it, generating path conditions similar to:

```
(tt[0.0105] + CONST_0.0016666666666666668) < tc[0.05] &&
(tt[0.0105] + CONST_0.00125) < tc[0.05] &&
(tt[0.0105] + CONST_8.333333333333334E-4) < tc[0.05] &&
(tt[0.0105] + CONST_4.166666666666667E-4) < tc[0.05] &&
(tt[0.0105] + CONST_0.0) < tc[0.05] &&
 tc[0.05] < tmst[50.0200005] &&
 tc[0.05] > tt[0.0105]
```

The above path condition is illustrated as generated by the choco constraint solver [13]. The inputs to the method that it provides solutions for are: *tt, tmst* and *tc*, representing the time at the end of a turn, the time of minimum separation after a turn, and the lookahead time (how far in future to look for conflicts), respectively. The proposed solution for each of these inputs is provided within square brackets immediately following the input name. That is, for *tt = 0.0105*, *tmst = 50.0200005* and *tc=0.05*, this condition becomes true, and the corresponding path will be executed. Notation *CONST_* simply represents the fact that the value that follows it is a constant.

We should be able to further modularize the code by extracting other parts into their own methods and applying SPF to each method individually. Of course,

the challenge would still remain in combining the individual analyses to generate inputs for the entire *Conflict_probe* method. A similar process would then need to be applied in order to eventually generate inputs for the TSAFE prototype itself. Compositional symbolic execution [15, 16] aims at addressing the problem of combining local solutions into solutions for larger pieces of code, but it is still an open research area that we intend to further investigate in the future.

4 Black-Box Test-Case Generation with JavaPathfinder

In this section, we motivate and explain the use of the JPF model checker for test-case generation based on the characteristics of the inputs to a program, and without taking into account its source code, i.e., we consider the program as a black box. In particular, we use the

Testing tools such as JUnit have become popular as they automate some testing steps. However, currently adopted tools primarily automate test execution and offer little support for test generation. The developers and testers often have good intuition to determine what tests should be generated, but must manually translate this intuition into actual tests. Such manual test generation is time-consuming, and often results in test suites that have poor quality and are difficult to reuse. This is especially the case for software that requires complex test inputs or operates on complex data structures.

Several approaches have been proposed in the literature for generating tests based on high-level descriptions of desired test suites. The key idea of such approaches is that testers specify their tests at a higher level of abstraction, and tools (e.g., the commercial T-VEC [17]) are then used to automatically generate concrete tests according to the specified ones. This approach has the potential to enable developers and testers to avoid manual test generation and focus on the creative aspects of testing and development.

For example, for the targeted separation assurance code, we need to generate coordinates for the primary and secondary airplanes involved, velocity, direction, and whether they are eligible for turns, as discussed in Section 2. At a high level, the developer should simply need to describe the ranges of values that these inputs can take, rather than writing code that generates their possible combinations. JPF provides support for such declarative descriptions of inputs to a program. For our system under test, for example, we developed the code that is illustrated in Listing 1.1. When this code is provided as input to JPF, JPF will systematically explore all the possible combinations of the specified input value ranges. The advantage is that the developer concentrates on specifying the inputs of interest for the particular problem, leaving the generic part of creating combinations to a tool that is designed for this purpose.

JPF has several features that facilitate test input generation in this fashion. In order to provide strong support for random generation of values from several domains (including integers, doubles, etc), the capability has been programmed in JPF to support declarative statements such as those illustrated on Lines 3, 9, 14, and 16. For infinite domains such as doubles, the user specifies ranges

of values. In our example, these values were specified by the developers of the algorithms. For example velocity of each plane ranges between 250 and 550 miles per hour, and we investigate all increments by 50 between these values (see Lines 14 and 17). An additional feature supported by JPF is the possibility, once ranges have been selected, to also create inputs for small perturbations of the selected values, for example to offset each velocity selected by 10 miles per hour on each direction. This can be very useful in assessing whether the current set of values is representative, i.e., by checking whether small perturbations affect the results obtained during testing.

```
1  double XA0 = 0, YA0 = 0;
2
3  double XB0 =getDouble(new double[] { -20, -15, -10, -5, 0,
       5, 10, 15, 20 });
4  double YB0 = getDouble(new double[] { -20, -15, -10, -5, 0,
       5, 10, 15, 20 });
5
6  double Xdistance = Math.sqrt((XB0*XB0 + YB0*YB0));
7  Verify.ignoreIf((Xdistance < 2 ) || (Xdistance > 20));
8
9  boolean A_eligible = Verify.getBoolean();
10 boolean B_eligible = Verify.getBoolean();
11 Verify.ignoreIf(!(A_eligible || B_eligible));
12
13 double PsiA0_deg = 0;
14 double vA = getDouble(new double[] { 250, 300, 350, 400,
       450, 500, 550 });
15
16 double PsiB0_deg = getDouble(new double[] { 0, 30, 60, 90,
       120, 150, 180, 210, 240, 270, 300, 330 });
17 double vB = getDouble(new double[] { 250, 300, 350, 400,
       450, 500, 550 });
18
19 boolean diverging = TestVector.isAircraftDiverging(XA0,YA0,
       PsiA0_deg, vA, XB0, YB0, PsiB0_deg, vB);
20
21 TestVector.addMethodToJUnitFile(XA0, YA0, PsiA0_deg, vA,
       XB0, YB0, PsiB0_deg, vB, A_eligible, B_eligible,
       diverging);
```

Listing 1.1. Input program for JPF to generate separation assurance test cases

Another key capability is related to specifying tests that are of no interest. Instead of generating a large number of tests, and subsequently filtering only interesting cases among those, JPF allows us to filter values that are of not interest during test generation. For example, Line 11 specifies that if neither of the two planes is eligible for a turn, it does not make sense to test the algorithm. Line 7 will reject all the tests where the two primary aircraft are too close or too far (less than 2 miles apart or more than 20 miles apart) initially, because

the algorithm developers are not interested in such cases. If the aircraft are less than 2 miles apart, for example, they are already in loss of separation (required minimum separation is 5 miles), so there is not much that the algorithm can do. If they are more than 20 miles apart, it is most likely that the problem will always be resolved. Although running a few tests for such values is a good idea in order to confirm the developers intuition about them, the main focus should be on the more challenging cases, as identified by them. The advantage of the proposed approach is that "*ignoreIf*" statements can easily be added or removed from the test specification, and the generated tests will be updated accordingly. Compare that to a process where the developer would have to explicitly program nested loops that generate combinations of values and exclude some values. The above specification makes the target tests clear, explicit, and easy to modify.

An advantage of filtering during the test input generation, as opposed to filtering after the fact, is efficiency. In our example, we have included a filtering statement at Line 7, immediately after selecting X and Y coordinates for the two primary aircraft. For the values that are thus eliminated, we avoid the generation of a very large number of combinations of those values with the potential values for velocity and direction of flight for the two aircraft. In fact, selecting good locations for the filtering statements can make a big difference in efficiency. Recently, a high-level language and an associated framework have been proposed that further facilitate the description and generation of tests [18]. The language, called UDITA, provides several high-level constructs for describing complex inputs and data structures. Through the implementation of features such as delayed choice of primitive values, it achieves improved efficiency while relying less on the quality of the test descriptions provided by the developer. We plan on experimenting with UDITA in the future.

5 Combinatorial Testing

As the full combinatorial exploration of all possible values of input parameter soon reaches infeasible numbers, we experimented with two alternative methods of testcase generation: Monte Carlo testcase generation and n-factor combinatorial exploration.

The Monte Carlo (MC) testcase generation treats each input variable as a statistical variable with a given probability density function, from which values for the test cases are randomly drawn. In most applications, a Gaussian distribution is assumed for all continuous inputs, whereby the mean usually is the nominal value. For discrete or discretized variables, a uniform probability distribution is assumed. In our TSAFE example, all variables are discrete, or have been discretized.

Monte Carlo (MC) test cases can be generated very easily. However, they provide no guarantee whatsoever regarding uniqueness and coverage of the input space. This means that for a reasonable coverage of the input space, a very large number of MC cases have to be executed, again quickly reaching the limits on what can be done practically. Even with a optimizations like discretization or

binning of variables, pre-filtering of test cases, or exploitation of domain-specific features, like symmetry, only an overall probabilistic measure on the coverage of the input space can be given. Thus, MC testcase generation can be used to quickly get an overview of the test space, and to provide a reasonably dense coverage in the vicinity of nominal conditions.

In real software programs, most errors are caused by a specific, single value of one input variable. The case that a fault is triggered by a specific combination of two variables is much less likely. Even more unlikely is the case that 3 input variables must have specific values in order to trigger the failure; the involvement of 4 or more variables can be, for most purposes, ignored. This observation (e.g., [19–21]) can be used to specifically tailor the generation of test cases, resulting in a substantially smaller number of test cases (see Table 2). Nevertheless, these cases completely cover all combination of variables up to a given bound n, hence the method is called n-factor combinatorial exploration.

No AC	No Vars	comb	1-fact	2-fact	3-fact	4-fact
2	5	$48 \cdot 10^3$	24	225 [<1s]	2,139 [1s]	15,095 [30s]
3	8	$87 \cdot 10^6$	34	459 [<1s]	6,047 [19s]	66,176 [3900s]
4	11	$160 \cdot 10^9$	34	612 [<1s]	9,784 [115s]	–
5	14	$294 \cdot 10^{12}$	34	709 [<1s]	12,940 [425s]	–

Table 2. Number of test cases generated with full exploration and n-factor exploration for TSAFE.

A number of efficient algorithms for the n-factor combinatorial testcase generation can be found in the literature (e.g., [22]). For the experiments in this paper, we used a tool developed at JPL [23] which extends the IPO algorithm [24]. Although the tool features a number of extensions, we only use the core algorithms to generate 3-factor combinatorial permutations for the discrete and discretized input variables of the TSAFE system. For testcase generation with continuous variables, this tool partitions the input space into discrete ranges, from which values are drawn randomly in a Monte Carlo fashion. For details see [23].

6 Extracting Requirements

As discussed in the introduction, a major challenge in testing separation assurance is the identification of oracles used for checking whether the tested code performs as expected. In other words, it is hard to express the conditions that characterize a successful versus a failed test. These conditions could be simple and generic, for example “no exception will be thrown”, but the more interesting ones are application specific and are those that capture whether the program performs its intended functionality. As mentioned, we call these “properties” or

"requirements". Finding and eliciting the requirements of a complicated piece of code like TSAFE is a non-trivial process.

We therefore took an interactive and iterative approach to extracting the requirements for TSAFE. In discussions with the algorithm and prototype developers, we first elicited some basic properties to test the code against. We started from rough, generic properties such as "none of the return values should be NaN". In fact, the code is supposed to compute some values in order to solve the loss of separation problem. When such values are "NaN", it means that the code did not attempt to compute them, which indicates a problem.

Through such basic tests, the algorithm developers got familiarized with our techniques, and we with their way of approaching the problem. We were therefore able to collaborate to gradually formulate more refined properties. This process often involved discussing properties that we all thought made sense, obtaining violations that were not meaningful, and subsequently realizing that a slightly modified or completely different property should be formulated instead. Some properties even emerged as a result of studying the results obtained when checking other properties. We will provide some examples in our presentation of the properties that follows.

We thus gradually created a set of requirements for the algorithms that is otherwise relatively hard to dig out of the code. These are also the requirements that the reference prototype will be tested against for robustness. These properties are generic, and should be applicable to other approaches of the loss of separation resolution problem.

Below is a list of the requirements that we have formulated so far and which the code has been tested against:

1. *The values that need to be computed by the algorithm should not remain at their initial value, i.e., NaN.*
2. *In the absence of secondary aircraft, if two planes are diverging then the algorithm should return no change in their direction of flight.* In general, a change of course to aircraft should only be advised if necessary for safety reasons. Therefore, even for aircraft that have lost separation, if their current courses are diverging, meaning that they are moving away from each other, then their direction of flight should not be changed.
3. *If a solution is returned for a specific bank angle, then loss of separation will also be avoided if the pilot uses a larger bank angle.* This is an important property. It is typical, when a change of course is advised with a specific bank angle, the pilot will slightly overcompensate on the bank angle when implementing the change. Proposing a solution that is not robust to increasing the bank angle would be dangerous.
4. *The minimum separation achieved by the solution proposed by the algorithm should always be larger than the minimum separation achieved if the aircraft were to remain on their course of flight.* This is a more generic version of property 2 about diverging aircraft.
5. *When only one / both primary aircraft are eligible for turning, the algorithm should solve all cases for which the time to first loss of separation is larger*

than 90 seconds, and 45 seconds, respectively. This property only applies in the absence of secondary aircraft, since in the latter case the position of the secondaries may always invalidate a good solution for the primaries. The property captures the intuition that, if the loss of separation is to occur far enough in the future, then there should be no cases that the algorithm could solve, meaning that the algorithm will be able to propose a course that will avoid loss of separation. Clearly, when the air traffic controller is able to propose a change of course for both aircraft involved, the algorithm could resolve cases for which the time to loss of separation is smaller (45 seconds, as opposed to 90 seconds). Note that our testing showed that the algorithms of our collaborators were able to resolve all cases for 85 seconds / 25 seconds, respectively.

6. *If the algorithm is able to resolve loss of separation for some input data, then it should be able to also resolve loss of separation for different, but symmetric input data.* This requirement emerged while checking property 5, where we were recording the number of tests for which the algorithm did not achieve the required minimum separation as a function of the time to first loss of separation. For the time to first loss of separation, we created bins of 5 second intervals, for example [5, 10) seconds, [10, 15) seconds, etc. Our outputs identified that for some cases the number of unresolved cases was odd when we knew that our tests generated with JPF were symmetric. This revealed a subtle bug in the code, and triggered the formulation of this property.
7. *If the algorithm is applied with secondary aircraft and extended turns enabled, then more cases should be resolved overall as compared to running the algorithm with extended turns disabled.* Extended turns allow the aircraft to perform two sequential maneuvers, where the first addresses the primary conflict, and the second addresses a potential conflict with a secondary aircraft that is caused as a result. That option should only exist if it offers an overall improvement.

Note that our collaboration with the algorithm developers spanned over a year. As described in more detail in section 8, all of the above properties were violated at different stages of the algorithm development, which resulted in newer and more robust versions of the code. Moreover, as the code gets expanded to add new features of the research of our collaborators, these requirements are used for regression testing to ensure that new versions of the algorithm preserve them.

7 Testing Framework for TSAFE

In this section, we describe the automated testing framework that we developed for TSAFE. For each combination of input data that is generated with the techniques described above, we automatically create corresponding test code in Java. Each test creates and initializes the required objects and attributes, and subsequently invokes the system under test.

The code is included in one or more JUnit files. JUnit is a simple framework for writing repeatable tests [25]. It provides support for developers to prepare tests for their code, including set up of targeted objects, test methods, and expected results. Several test methods can be included in a single JUnit file. On execution, JUnit will apply all the methods and report statistics like how many tests have passed and how many have failed. Since our framework may potentially generate tens or hundreds of thousands of test methods (one for each combination of input values), we automatically partition the inputs of sets of at most 1000 to be included in each JUnit file.

After we started experimenting with secondary aircraft, our test suites grew to contain tens of millions of cases. For those large test cases, JUnit proved to be slow. For this reason, we wrote our own code for running the tests, and in order to take advantage of our multi-core machines, also implemented a multi-threaded approach to running the tests. Our tests ran twice as fast on a four core machine using the parallel version, as compared to the non-parallel one.

```
1  public static void checkError(...) {
2    if (!filesCreated) {
3       initializeFiles();
4       filesCreated = true;
5    }
6    recordTestCase(R, aircraftInputs, diverging, ttlos, ttNmac);
7    testsRun++;
8
9    String err = "";
10   err +=(checkAchievedRelativeSeparation(R, aircraftInputs)?
11            "" : " checkAchievedRelativeSeparation");
12   err += (checkForNaN(R, aircraftInputs)?
13            "" : " checkForNaN");
14   err += (checkForWarns(R, aircraftInputs)?
15            "" : " checkForWarns");
16   err +=(checkTurns(R, aircraftInputs, diverging)?
17            "" : " checkTurns");
18   err +=(checkAchievedAbsoluteSeparation(R, aircraftInputs,
19            ttlos)? "":" checkAchievedAbsoluteSeparation");
20   err += (checkAchievedNmac(R, aircraftInputs, ttNmac)?
21            "" : " checkAchievedNmac");
22   if (TestingConfig.secondaries > 0)
23        err += (checkSecConflicts(R, aircraftInputs)?
24            "" : " checkSecConflicts");
25
26   Assert.assertTrue(err + " " +
27      aircraftInputs, "".equals(err));
28 }
```

Listing 1.2. Application of oracles to tested code

Our testing framework also implements the requirements discussed in Section 6. We coded these properties separately from the prototype code so that they

can be reused in implementations of other separation assurance algorithms. As illustrated in Listing 1.2, all the properties that we check during our testing are included in a single method, which subsequently invokes the right method for each property. For example, Listing 1.3 encodes Requirement 2, according to which—when aircraft are diverging—the algorithm should advise no change in course. As part of each requirement, we code a message that should be returned if that requirement gets violated, and a separate file in which this message will get printed, e.g., `out[Turn]` in Listing 1.3. In this way, as a result of running our tests, we obtain several files, each file documenting all the test inputs that violated the requirement associated with the file, and a message explaining the problem. This feedback has proven invaluable for the developers of the code in debugging their algorithm when errors were discovered.

```
1  public static boolean checkTurns(...) {
2    if (diverging) {
3      if (R.A_best_turn_ang != 0 || R.B_best_turn_ang != 0) {
4        out[Turn].println("Found error in test case:\n    "
5          + aircraftInputs);
6        out[Turn].println("Turn angle for aircraft A: "
7          + R.A_best_turn_ang);
8        out[Turn].println("Turn angle for aircraft B: "
9          + R.B_best_turn_ang);
10       out[Turn].println(
11         "Not an optimal solution for diverging aircraft");
12       out[Turn].flush();
13       return false;
14     }
15   }
16   return true;
17 }
```

Listing 1.3. Checking requirement for diverging aircraft

Note that for our testing to be independent of the prototype code, we have implemented our own methods for calculating whether aircraft are diverging, the time to first loss of separation, etc. If we were to use the implementations in the prototype, our error checking code could be susceptible to the same errors as the system under test, which could interfere with the error-finding capabilities of our framework.

8 Results

In this section, we discuss our test results and use them in order to compare the advantages and disadvantages of the proposed approaches. We use code coverage as a measure of the effectiveness of the different approaches, but also discuss their error finding capabilities.

Code coverage. We compared the different testcase generation techniques for the case of 2 primary and one secondary aircraft. Code coverage was analyzed

using the tool CodeCover [26]. This open-source tool can be used from command-line or as an Eclipse plug-in. It measures statement coverage (each basic statement must be covered), branch coverage (all branches of conditional statements must be executed), loop coverage, and strict condition coverage. The loop coverage metric of CodeCover analyzes which loop has been executed zero times (i.e., the body has not been entered), once, or multiple times. Finally, strict condition coverage exercises all combinations of the Boolean subexpressions.

We focus on the *Conflict_resolution* and *MultiConflict_resolution* classes, since these implement the actual conflict detection and resolution. Test suite T_1 has been generated by exhaustive combinatorial exploration with JPF, including the filtering mechanism used to avoid generation of uninteresting tests, as discussed in Section 4. Test suite T_2 has been generated using 3-factor combinatorial exploration, using the same input ranges for the variables. This test suite contains 6,047 test cases. T_2, however, has not taken advantage of any of the domain-specific filtering. For this reason, we have additionally produced the set T_2^f, which contains all test cases from T_2, which pass the domain-specific filters. Finally, T_3 is a set of 6,047 traditional Monte Carlo test cases. Please note that for all test suites, the same discretization of input variables and variables is being used. T_3^f are the 698 test cases obtained from T_3 by applying the filters.

Test-suite	No tests	Coverage: *Conflict Res.*				Coverage: *MultiConflict Res.*			
		Stmt	Branch	Loop	Strict Cond	Stmt	Branch	Loop	Strict Cond
T_1	$9.9 \cdot 10^6$	94	90	46	85	94	92	37	83
T_2	6047	93	89	46	83	94	91	37	81
T_2^f	712	92	89	46	83	63	57	32	52
T_3	6047	93	89	46	83	64	57	30	52
T_3^f	698	92	89	46	83	63	57	30	52

Table 3. Coverage Results

Table 3 shows the coverage results, given as a percentage. Note that 100% coverage may not be achievable due to unreachable code. However, for this study, we are interested in relative, rather than absolute, coverage achieved by the various approaches.

The coverage achieved for the conflict resolution module does not exhibit significant differences when tested with the different test suites. However, there are some subtle differences, where T_1 executes a few lines of code that the other test suites miss. In examining statement coverage, for example, we observed that the main difference between T_1 and T_2 coverage is a special case ("no local maximum during turn, use absolute maximum"), which obviously needs a special setting of multiple input variables, namely a specific setting of the speed of both aircraft, a heading of the second aircraft, which is correlated to the speed, and $xB0 = \pm 5$.

For the particular instance of checking the algorithm with two primary and one secondary aircraft, the developers requested that we generate only test cases for which both primary aircraft are eligible for turns. As a result, none of our test suites was able to achieve 100% statement coverage. For example, the code shown in Listing 1.4 was not covered, which was to be expected.

```
1  if ((xB0 > 0 && psiB > 0 && psiB < Math.PI) ||
2      (xB0 < 0 && psiB > Math.PI && psiB < 2.0 * Math.PI)) {
3                                  maneuver = false;
4                                  return; }
```

Listing 1.4. Java code not covered by the test suites

The most striking difference is a substantial drop in all types of coverage when the 3-factor test suite T_2 is subjected to the domain-specific filters. Since the 3-factor test case generation is already highly optimized with respect to its coverage, removing any of those results in a big loss of code coverage. In the case of the module *MultiConflict_resolution*, this causes the code that handles extended turns (several hundred lines) to never be executed.

Discussion. It is clear from this small experiment that the number of test cases when using combinatorial testing drops by several orders of magnitude as compared to the exhaustive approach based on JPF. An advantage of the JPF approach is the clear declarative description of input ranges, of the domain-specific filters to be applied to them, as well as of potential perturbations of the input values used for checking the quality of the generated tests. Filtering during generation also increases efficiency since it is able to prune out early subtrees of combinations rooting at uninteresting values. Our code coverage results also showed that the JPF based approach is able to achieve significantly better coverage than combinatorial approaches when restricted to the test inputs of interest to the developers (meaning only filtered test cases).

As discussed in Section 6, our testing identified violations of all the stated requirements at different stages of the code development. For example, when we ran our experiments with and without extended turns, we discovered that the results were the same, showing that the code that was enabling extended turns was not exercised. When this error was fixed, we discovered that for many cases the results were better when extended turns were disabled. Although this can happen sometimes, it was happening more often than expected, which again identified a subtle error in the prototype. Finally, the fact that symmetric input cases would achieve different results uncovered another bug.

A measure of the quality of a test suite is also its error finding capabilities. From the cases discussed above, it was clear that it is worthwhile applying the exhaustive JPF based approach while the resulting test suites are manageable. Combinatorial test suites for example are not necessarily symmetric, so the related problem could not have been discovered. Similarly, the developers asked us to generate diagrams that illustrate some of our obtained results, and which they inspected to see if they matched their intuitions. One such diagram displays the achieved minimum separation as a function of the time to conflict between

the primary aircraft. In particular, they were interested, for each value of the time to conflict, in illustrating the test case that achieved the highest, and the test case that achieved the lowest minimum separation. There was a clear point in that diagram that would stand out as not meaningful, which resulted in the developers updating the prototype code. A large number of tests increases the possibility of identifying such points in diagrams such as these. On the other hand, when multiple secondary aircraft come into the picture, clever techniques such as combinatorial testing are necessary. Our expectation is that an incremental approach starting with exhaustive and continuing with more selective test generation techniques, will be effective in ensuring a robust prototype for our targeted algorithms.

9 Statistical Analysis of Test Results

The large number of test cases that the presented techniques generate are used to thoroughly exercise the software under test. For each test run, values of output variables are recorded, and the results are checked against requirements. In the case of our TSAFE example, output variables include discrete information, like the best resolution strategy (e.g., "AC A left, AC B right"), as well as continuous information like the minimal separation between the two aircraft in nautical miles or the time to loss of separation. In our case, a total of 13 variables are recorded.

As mentioned earlier in the paper, formulating requirements for separation assurance is a challenging task. Even though we have spent a lot of effort in identifying requirements, we are nowhere close to having a complete set of requirements for the problem of separation assurance. We therefore also investigate an orthogonal and complementary approach to evaluating the results of the testing process. More specifically, we try to identify anomalies by extracting information from the testing process as the test results can be considered to be a large, high-dimensional data set.

Statistical single variable analysis looks at each of the output variables in isolation. This can provide valuable information about expected and actual value ranges as well as outliers, which could indicate a software error. However, much more information about the behavior of TSAFE and, in particular, erroneous behavior, could be deduced from the interaction of multiple output variables and the analysis of their correlation.

For this experiment, we have used clustering, a well-known technique to automatically find structure in large, multivariate data sets. Clustering is an unsupervised learning method that tries to estimate the class membership matrix and class parameters, only given the data. We are using the AutoBayes tool to automatically generate tailored clustering algorithms. AutoBayes [27, 28] is a fully automatic program synthesis system, developed at NASA Ames that generates efficient and documented C/C++ code from abstract statistical model specifications. From the outside, AutoBayes looks similar to a compiler for a very high level programming language: it takes an abstract problem specification

in the form of a (Bayesian) statistical model and translates it a customized and documented algorithm in C that can be called from Matlab to process the actual data (other target platforms are supported as well).

On the inside, however, AutoBayes works quite differently: AutoBayes first derives a customized algorithm skeleton implementing the model and then transforms it into optimized C/C++ code (for Matlab and Octave [29, 30]). Hereby, the input specification is translated into a Bayesian Network [31]. The program synthesis system uses a schema based approach to break down large problems into statistically independent subproblems and tries to solve them symbolically. If no solution can be found, a customized numerical algorithm is instantiated. The synthesis task is heavily supported by a domain-specific schema library, an elaborate symbolic subsystem, and an efficient rewriting engine. For details on AutoBayes see [27].

In order to perform clustering, the statistical distribution or probability density function for each variable should be known. For TSAFE, several variables have discrete values (booleans or enumeration types), which have a discrete distribution; continuous variables include distances, times, and angles. Most clustering algorithms and tools make the assumption that all variables are Gaussian (normal) distributed and Gaussian noise is added to the discrete variables in order to gain a Gaussian-like distribution. However, such a model can be statistically very inaccurate, in particular, when dealing with angles. A noisy measurement of an angle close to 0° would, when considered as Gaussian distributed, yield two classes with means around 0° and 360°.

In our TSAFE example, we use a vonMises [32] distribution for the angles (e.g., `best_heading_angle`), Gaussian distribution for the other continuous variables, and a discrete probability density function for the discrete variables.

From such a compact specification of less than 100 lines, AutoBayes generates several thousand lines of documented C code with a Matlab MEX interface. Internally, an EM (Expectation Maximization) algorithm [33], an iterative statistical optimization algorithm is generated. In our case, all statistical subproblems could be solved symbolically and code as well as a detailed derivation is generated within a few seconds of run time. For examples of specifications and autogenerated, detailed derivations we refer the reader to the AutoBayes manual [28].

If necessary, this specification can easily extended to incorporate additional variables and domain knowledge in the form of priors. Again, code can be generated automatically without the user having to specify details about the desired algorithms. Generally available EM-implementations, like Autoclass [34], EMMIX [35], or MCLUST [36] could be used for clustering. These algorithms are usually designed for Gaussian distributed data only and are thus not useful for the task at hand. Refining the statistical model (e.g., by incorporating other probability distributions for certain variables or to introduce domain knowledge), the EM-algorithm needs to be modified substantially for each problem variant, making experimentation a time-consuming and error-prone undertaking with such tools.

Application to TSAFE. The results of the tests are huge, high-dimensional data sets with 13 variables. We use an AutoBayes-generated clustering algorithm to automatically detect structure in this data set. A good clustering result could be obtained by separating the data into 5 different classes. Here, one class contains around 60% of the data, which comprise the nominal case, i.e., there is no conflict to be resolved. One other class picks up most test cases, where the minimal separation is less than the required 5 nautical miles, and which also use specific methods for conflict resolution. However, the classes are not separated by values of a single variable, indicating that the behavior of the system cannot be determined by a single variable only. Hence, a single variable analysis will not reveal important details.

For visualization of the clustering results, we use scatter plots, i.e., plots showing the covariance between two variables. Figure 3 shows an example (using visualization routines of the NASA Ames margins tool). The value of the output variable "best_turn_dirs" is shown over the x coordinate for the aircraft B for each test case. In the output, each data point is colored according to which of the 5 classes it belongs. A brown color corresponds to the nominal class. For members of this class, no turn maneuver has to be carried out. In order to make the number of cases for which a specific value occurs and the class memberships better visible, we perturb all of these discrete values. The larger the blob, the more data points have this value.

The TSAFE domain is highly symmetric. As discussed above, the algorithm and all test cases are symmetric with respect to the `X` coordinates. Thus, one can expect that the output values will be symmetric as well. However, some scatter plots did, for an early version of the software, not show a symmetric behavior (Figure 3). Using the color (class) information, potential sources of errors can be located. Although this non-symmetric behavior directly corresponds to a coding error in the early program version, such analysis approach cannot provide rigorous evidence for an error or absence of error. Rather, the statistical clustering analysis of test results can be seen as an additional support for the designer and program analyst. AutoBayes can also support sensitivity analysis and help in finding safety margins, i.e., ranges of input parameters, where failures are likely to occur. If these are not as expected, a flaw in coding of design might have been detected.

10 Conclusions and Future Work

This paper reported on our experience with applying several formal approaches for testing separation assurance algorithms. We presented the various techniques in detail, and discussed their relative strengths and weaknesses. We believe that our experience could be valuable in verifying other problems in the aerospace domain.

We advocate an incremental approach: we start from generating tests in an elegant way using JPF to exhaustively generate combinations of inputs that have been described in a concise, declarative fashion. Domain-specific filters are also

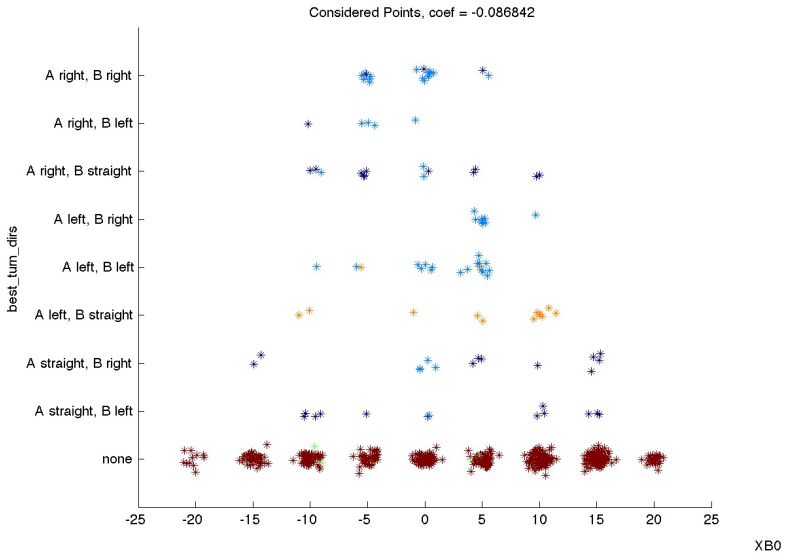

Fig. 3. Scatterplot over test results. An early version of the program exhibited a non-symmetric behavior with respect to the X axis (XB0), a potential indicator of an error.

applied to focus the tests as specified by the algorithm developers. Moreover, we start from the statement of generic requirements, that gradually get refined to capture more detailed properties of the targeted system. As more components are introduced into the problem (for example secondary aircraft), combinatorial test case generation techniques are used to generate more restricted test suites of manageable size. Finally, we propose to analyze the large number of data generated during the testing process with clustering approaches that may provide indications of irregularities in the results, and which may trigger the formulation of additional requirements.

More sophisticated test case generation techniques like symbolic execution were not able to routinely handle separation assurance code. In the future, we plan to investigate a combination of our current exhaustive test case generation approach with symbolic execution, as a way of increasing the power of the former, and enabling the application of the latter for aerospace code. Moreover, we intend to investigate compositional symbolic execution techniques [15, 16], in order to address the scalability issues that we experienced.

Acknowledgements. We wish to thank Heinz Erzberger and Karen Heere for their constructive advice on their algorithms and code, as well as their overall collaboration on this exciting project. We also gratefully acknowledge Peter Mehlitz for his help with JavaPathfinder, and Todd Farley and the NextGen program for funding this work.

References

1. D. H. Bushnell, D. Giannakopoulou, P. Mehlitz, R. Paielli, and C. Pasareanu, "Verification and validation of air traffic systems: Tactical separation assurance," in *Proc. IEEE Aerospace Conference*. IEEE Press, 2009.
2. H. Erzberger and K. Heere, "Algorithm and operational concept for resolving short-range conflicts," *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 224, no. 2, pp. 225–243, 2010.
3. RTCA, "Do-178b: Software considerations in airborne systems and equipment certification," 1992. [Online]. Available: http://www.rtca.org
4. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.
5. "Javapathfinder." [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf
6. E. Model Checking Clarke and D. Grumberg, O.and Peled, *Model Checking*. MIT Press, 2000.
7. N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Trans. Software Eng.*, vol. 20, no. 9, pp. 684–707, 1994.
8. A. Betin-Can, T. Bultan, M. Lindvall, B. Lux, and S. Topp, "Application of design for verification with concurrency controllers to air traffic control software," in *ASE*, 2005, pp. 14–23.
9. M. Consiglio, V. Carreno, D. Williams, and C. Munoz, "Conflict prevention and separation assurance method in the small aircraft transportation system," *Journal of Aircraft*, vol. 45, no. 0021-8669, pp. 353–358, 2008.
10. G. Dowek and C. Munoz, "Conflict detection and resolution for 1,2,...,N aircraft," in *Proceedings of the 7th AIAA Aviation, Technology, Integration, and Operations Conference*, 2007.
11. J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.
12. C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *ISSTA*, 2008, pp. 15–26.
13. "The choco constraint solver." [Online]. Available: http://choco.sourceforge.net/
14. IASolver. [Online]. Available: http://www.cs.brandeis.edu/ tim/Applets/IA-solver.html
15. P. Godefroid, "Compositional dynamic test generation," in *POPL*, 2007, pp. 47–54.
16. P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali, "Compositional may-must program analysis: unleashing the power of alternation," in *POPL*, 2010, pp. 43–56.
17. "T-vec." [Online]. Available: http://www.t-vec,com
18. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *32nd International Conference on Software Engineering (ICSE)*, 2010.
19. D. Cohen, S. Dalal, J. Parelius, and G. Patton, "The combinatorial design approach to automatic test generation," *Software, IEEE*, vol. 13, no. 5, pp. 83–88, Sep 1996.
20. I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, "Applying design of experiments to software testing: experience report," in *ICSE '97: Proceedings of the 19th international conference on Software engineering*, 1997, pp. 205–215.
21. D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: an analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering*, vol. 8, no. 4, 2001.

22. S. F. A. Mats Grindal, Jeff Offutt, "Combination testing strategies: a survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, pp. 167–199, 2005.
23. J. Schumann, K. Gundy-Burlet, C. P. X. Xareanu, T. Menzies, and T. Barrett, "Software v&v support by parametric analysis of large software simulation systems," in *Proc. IEEE Aerospace*. IEEE Press, 2009.
24. K. Tai and Y. Lie, "A test generation strategy for pairwise testing," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 109–111, 2002.
25. "Junit." [Online]. Available: http://www.junit.org/
26. "Codecover—an open-source glass-box testing tool," 2009. [Online]. Available: http://codecover.org
27. B. Fischer and J. Schumann, "AutoBayes: A system for generating data analysis programs from statistical models," *J. Functional Programming*, vol. 13, no. 3, pp. 483–508, May 2003.
28. J. Schumann, H. Jafari, T. Pressburger, E. Denney, W. Buntine, and B. Fischer, "Autobayes program synthesis system users manual," NASA, Tech. Rep. NASA/TM-2008-215366, 2008.
29. M. Murphy, "Octave: A free, high-level language for mathematics," *Linux Journal*, vol. 39, Jul. 1997.
30. "Octave," 2010. [Online]. Available: http://www.gnu.org/software/octave
31. W. L. Buntine, "Operations for learning with graphical models," *J. AI Research*, vol. 2, pp. 159–225, 1994.
32. C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 1st ed. Springer, 2006.
33. A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm (with discussion)," *J. of the Royal Statistical Society series B*, vol. 39, pp. 1–38, 1977.
34. P. Cheeseman and J. Stutz, "Bayesian classification (AutoClass): Theory and results," in *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining*, U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, Eds. AAAI Press, 1996, pp. 153–180.
35. G. McLachlan, D. Peel, K. E. Basford, and P. Adams, "The EMMIX software for the fitting of mixtures of normal and t-components," *J. Statistical Software*, vol. 4, no. 2, 1999.
36. C. Fraley and A. E. Raftery, "MCLUST: Software for model-based clustering, density estimation, and discriminant analysis," Department of Statistics, University of Washington, Tech. Rep. 415, Oct. 2002.