Software Validation via Model Animation

Aaron M. Dutle, César A. Muñoz, Anthony J. Narkawicz, and Ricky W. Butler

NASA Langley Research Center, Hampton, Virginia 23681-2199

Abstract. This paper explores a new approach to validating software implementations that have been produced from formally-verified algorithms. Although visual inspection gives some confidence that the implementations faithfully reflect the formal models, it does not provide complete assurance that the software is correct. The proposed approach, which is based on animation of formal specifications, compares the outputs computed by the software implementations on a given suite of input values to the outputs computed by the formal models on the same inputs, and determines if they are equal up to a given tolerance. The approach is illustrated on a prototype air traffic management system that computes simple kinematic trajectories for aircraft. Proofs for the mathematical models of the system's algorithms are carried out in the Prototype Verification System (PVS). The animation tool PVSio is used to evaluate the formal models on a set of randomly generated test cases. Output values computed by PVSio are compared against output values computed by the actual software. This comparison improves the assurance that the translation from formal models to code is faithful and that, for example, floating point errors do not greatly affect correctness and safety properties.

1 Introduction

The formal verification of software written in widely used programming languages such as Java and C++ faces many hurdles. A typical approach for developing safety-critical software in these languages consists of specifying and verifying the critical components of the software as algorithms in a formal verification system, and then, translating either automatically or manually these formal models into code. In this approach, visual inspection and peer-review techniques are used to provide some assurance that the implemented code faithfully reflects the formal models. However, despite the best efforts, implementation errors can be accidentally introduced during the translation process.

The difficulty of the typical approach is increased by the large semantic gap that exists between modern programming languages and the functional specification languages often used in formal models. For example, imperative languages support control structures for iteration that must be cast as recursive functions in functional specification languages. This is complicated by the fact that iterations in modern languages may produce side effects on an arbitrary number of variables within their scope. In embedded systems, some of these complications are avoided by restricting the programming languages to certain constructs.

However, for convenience and efficiency reasons, enforcing such restrictions is not always desirable or even possible. Another difficulty arises from the fact that modern programming languages utilize floating point arithmetic while formal verification is usually performed over the real numbers. Therefore, bridging the gap between implementations and their formally verified counterparts is a challenging problem in the validation and verification of critical software.

Significant value can be obtained by validating the numerical computations of a program against the actual theoretical values. Many subtle errors in the specification and implementation of an algorithm can be discovered and repaired by this process. For example, numerical errors can cause the software to make completely different decisions from what would be done if the computations were performed using exact values. The authors have found cases where two different implementations of a formally-verified conflict resolution algorithm [20], computed resolution maneuvers in opposite directions. This occurred even though the two implementations, one in Java and the other in C++, were syntactically almost identical. This undesirable behavior was due to the Java and C++ compilers producing a different order of evaluation of an expression, which resulted in different floating point results.

This paper explores a practical approach to the validation of software that implements formally-verified algorithms. The approach, which is called *model animation*, is based on animation[1] of formal specifications. The technique compares computations performed in the software implementations against those symbolically evaluated on the corresponding formal models. While model animation does not provide an absolute guarantee that the software is correct, it increases the confidence that the formal models are faithfully implemented in code. The proposed approach is illustrated on a library of kinematic software used for trajectory generation in conflict detection and resolution algorithms. The validated library, which implements formally-verified algorithms, is one of the core components of a prototype software for aircraft separation assurance. This prototype software is under development at NASA Langley and is being used for fast time simulations of advanced air traffic management (ATM) concepts.

2 Model Animation

In this paper, the concept of *software validation* refers to the process of checking that a software component meets its formal specification. The proposed software validation approach assumes the availability of formally-verified models of the software's critical algorithms in the specification language of an interactive theorem prover. It also assumes that the software implementations follow the

[1] The term animation used here refers to having a (usually static) specification actually perform calculation. In this sense, the formal model is brought to life, or *animated*. This is not to be confused with a tool such as PVSioWeb [16] which provides a graphical interface to, and interaction with, a PVS specification.

control and data structures of the formal models[2]. These two assumptions can be satisfied by either manual or automatic translation [13]. Furthermore, they do not have to be satisfied in any order. Indeed, an advantage of the proposed approach with respect to the correct-by-construction approach [17] is that formal models can be written a posteriori, which is usually done in the validation of legacy critical code. The model animation technique involves the following steps.

1. Automate the calculation of exact answers for specific inputs of the formal model. Where exact answers are not possible, e.g. formulas involving transcendental functions, provide semantic attachments that enable precise computations on the formal models.
2. Automatically generate input values and compare the symbolic evaluation of these values in the formal models to those computed by the software implementation, to determine if are equal up to a specified tolerance.

This approach is illustrated on core component of a prototype air traffic management (ATM) software package called Stratway, which is being developed at NASA Langley [10]. Stratway provides conflict detection and resolution algorithms using kinematic aircraft trajectories. These trajectories are generated in Stratway from a flight plan described by a sequence of 4D waypoints (aircraft position and time). The simplest model for flight based on this flight plan would be to assume that an aircraft follows a straight line trajectory with constant velocity between each successive pairs of waypoints. Of course, an aircraft cannot actually fly such a model consistently, since all but the most basic flight plans contain instantaneous changes in velocity and direction. On the other hand, high-fidelity modeling of how an aircraft would actually fly a given flight plan is both dependent on the dynamics of the aircraft, and the details of its control systems. The trajectories generated in Stratway strike a balance between these two extremes, producing trajectories with continuous velocities that obey the basic laws of motion. Instantaneous changes in direction are replaced with circular arcs, and instantaneous changes in ground or vertical speed are replaced with segments of constant acceleration [11]. The resulting kinematic flight plan is a sequence of points (called trajectory change points, or TCPs) where each segment between successive points is either 1) a constant velocity straight line segment, 2) a constant vertical acceleration segment, 3) a constant ground speed acceleration segment, or 4) a circular turn segment. This kinematic flight plan is compact in its representation and also gives a realistic picture of an aircraft flying a given route.

The functions that compute the position and velocity of an aircraft throughout each type of segment, as well as functions for determining the amount of time needed to keep the velocity continuous throughout the flight plan, reside in a kinematics library used by Stratway. Much of the core functionality of Stratway,

[2] While this assumption is not strictly necessary for the approach to be carried out, this *syntactic similarity* is one reason for trusting the software implementation. The *behavioral similarity* justified by the outlined approach provides the other.

including trajectory generation and conflict detection and resolution algorithms, depends on the correctness of this library. Hence, strong assurance of the correctness of these basic kinematic functions is desired for a safety-critical application. On the other hand, Stratway is intended to be used as a convenient tool for simulation, and for testing new algorithms and concepts for air traffic management. Because of this, Stratway is available in both Java and C++ software libraries. The formal verification of the actual code is extremely challenging, which is why a practical approach to validate the software components of the library against their formal models was undertaken.

For the ATM software examined, the core algorithms used in the kinematic library were formally specified and verified in the Prototype Verification System (PVS) [21]. The formal verification of these algorithms involved several aspects. Foundational theorems are proved showing that the algorithms used for position and velocity obey basic Newtonian physics. For example, a function computing a velocity based on acceleration is proven to be equal to the integral of the acceleration. Putative theorems are also proven in the theorem prover to show the algorithms perform their desired task. For instance, for an algorithm designed to model an aircraft moving from its current altitude to a target altitude, one such theorem would say that the altitude at termination of the algorithm *is* the target altitude.

An assumption of the proposed approach is that the software implementations and the formal models share similar data and control structures. Ideally, variable and function names should be preserved. However, this is not always possible due to different naming conventions in the languages involved. The syntactic similarity allows for a simpler visual comparison of the different versions of the algorithms, which increases the confidence that they do the same computation. For the kinematic ATM software, the PVS formal models were manually translated into Java and C++ code. This paper focuses on the Java code, but the same approach can be used on the C++ code. Much of the kinematic ATM software analyzed in this project already existed, and the formalization was done to give a higher level of assurance of its correctness. For a few of the existing algorithms, the formal specification and putative theorems revealed subtle errors, which were subsequently corrected.

Algorithms written in functional specification languages, such as PVS, cannot always be evaluated due to the presence of non-computable operations over real numbers such as square root and trigonometric functions. This issue is addressed in the proposed approach by providing semantic attachments [9] that compute guaranteed approximations of real-valued functions. In the case of PVS, the animation of functional specifications, including semantic attachments, is supported by the animation tool PVSio [19]. PVSio provides semantic attachments for several real number functions that are guaranteed to be correct up to a given precision. These semantic attachments do not guarantee that all computations are correct up to that precision, as approximation errors accumulate, but they significantly improve the quality of numerical outputs over floating point computations.

One important aspect of the proposed approach is to determine an appropriate collection of test input values for each algorithm. The following process is used. First, for each parameter of an algorithm, an appropriate range for the parameter is determined. For example, an altitude parameter is restricted to be between 0 and 40,000 feet. Three models for testing are then used. In the first model, a sequence of inputs are randomly selected to lie within the specified ranges for the parameters, and the software and PVS output compared, checking to see if they are the same up to a defined tolerance. In the second model, the range of each parameter is split according to a mesh size. For example, the altitude parameter might be split into 1000 foot blocks. For an algorithm with N different inputs, this splits the input space into an N dimensional grid, and the software and PVS outputs are compared at each intersection point. The third method starts with the same grid as in method two, but instead of testing at the grid intersection points, a random point from inside each block that this grid defines is selected for comparison of the software and PVS outputs. For methods two and three, variation in the mesh size allows for a tradeoff between the level of assurance that the software and PVS algorithms agree and the amount of time and computer resources used.

3 Formalization and Implementation of ATM Kinematic Library

Among the several algorithms comprising Stratway's library, the algorithms that were validated using the proposed technique are those related to the generation of kinematic trajectories and, in particular, the algorithms that deal with turn dynamics, vertical acceleration, and ground speed acceleration. Furthermore, in order to complete the full specification, a host of additional helper algorithms and datatypes had to be specified. For example, a large collection of basic vector operations were implemented, including projections between 3D and 2D vectors, conversions to and from velocity vectors specified in Euclidean coordinates versus vectors in polar coordinates specified by track angle and ground speed, and many others.

In addition to specifying the algorithms that are used explicitly in the kinematics library, a wide variety of mathematical background must be built into the theorem prover in order to prove the foundational and putative theorems that provide assurance that the algorithms are specified correctly. For instance, in order to prove that a velocity function is the integral of a specified acceleration, the theory of integration must be accessible to the theorem prover. The NASA PVS Library[3] contains much of the required mathematical background (including integral calculus [6]), but the required mathematics is almost never fully ready to apply directly. For example, the vertical speed algorithms are essentially described by piecewise constant acceleration functions. In order to prove the corresponding foundational theorems, a theory of piecewise defined functions and their integration was written and employed.

[3] `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library`.

For brevity, the remainder of the section focuses on the case of turn dynamics.

3.1 Turn Dynamics in PVS

The kinematics library includes algorithms to compute the trajectory of an aircraft in a frictionless banked turn, which is turning to leave one leg and join another leg of a predetermined flight plan. The trajectory of such an aircraft traces out a circular arc. In PVS, algorithms are specified as strongly-typed functions. For instance, the following PVS function computes the position and velocity of a turning aircraft at a given time.

```
turnOmega($\mathbf{s}_o, \mathbf{v}_o$: Vect3, $t$: real, $\omega$: real): [Vect3, Vect3] $\equiv$
    IF  $\omega = 0$ THEN  $(\mathbf{s}_o + t \cdot \mathbf{v}_o, \mathbf{v}_o)$
    ELSE LET
        $v = \texttt{groundSpeed}(\mathbf{v}_o)/\omega$,
        $\mathbf{s} = (\mathbf{s}_{ox} + v \cdot (\cos(\texttt{trk}(\mathbf{v}_o)) - \cos(t\omega + \texttt{trk}(\mathbf{v}_o)))$,
            $\mathbf{s}_{oy} - v \cdot (\sin(\texttt{trk}(\mathbf{v}_o)) - \sin(t\omega + \texttt{trk}(\mathbf{v}_o)))$,
            $\mathbf{s}_{oz} + t\,\mathbf{v}_{oz})$,                                          (1)
        $\mathbf{v} = (\texttt{groundSpeed}(\mathbf{v}_o) \cdot \sin(t\omega + \texttt{trk}(\mathbf{v}_o))$,
            $\texttt{groundSpeed}(\mathbf{v}_o) \cdot \cos(t\omega + \texttt{trk}(\mathbf{v}_o))$,
            $\mathbf{v}_{oz})$ IN
        $(\mathbf{s}, \mathbf{v})$
    ENDIF .
```

The parameters $\mathbf{s}_o$ and $\mathbf{v}_o$ are vectors in $\mathbb{R}^3$ that represent the initial position and velocity of the aircraft, respectively. The parameter t is the future time at which the state of the aircraft along its turn is computed. Finally, ω is the angular velocity. The output of this function is the position and velocity of the aircraft at the time t along its turn, which is relative to the current time. The function `trk` used here computes the track angle of a vector as measured from true north.[4] For a banked turn, there is a simple relationship between the angular velocity ω and the radius R, given by the following equation.

$$\omega = \texttt{dir} \cdot \texttt{groundSpeed}(\mathbf{v}_o)/R.$$

The parameter `dir` is either -1 or 1, depending on whether it is a right turn or a left turn, respectively.

The following theorem expresses the correctness of the function `turnOmega`. It states that for all times t, the distance between the position output of `turnOmega` and the center of the turn is given by the turn radius.

[4] As typical in air navigation, angles are measured clockwise with respect to true north.

Theorem 1. *For all* $t \in \mathbb{R}$*,* $\mathbf{s}_o, \mathbf{v}_o \in \mathbb{R}^3$*,* $\omega \neq 0 \in \mathbb{R}$*, let* $v = \frac{\mathit{groundSpeed}(\mathbf{v}_o)}{\omega}$*,* $\mathbf{w} = \mathbf{s}_o + (v\,\cos(\mathit{trk}(\mathbf{v}_o)), -v\,\sin(\mathit{trk}(\mathbf{v}_o)), t\,\mathbf{v}_{oz})$*,* $(\mathbf{s}, \mathbf{v}) = \mathit{turnOmega}(\mathbf{s}_o, \mathbf{v}_o, t, \omega)$*, then*

$$\|\mathbf{s} - \mathbf{w}\| = |v|.$$

Theorem 1 implicitly states that **w** is the center of the turn. This theorem has been formally proved in PVS and its proof depends only on basic properties of sine and cosine. Figure 1 illustrates the geometric relations involved in Theorem 1.

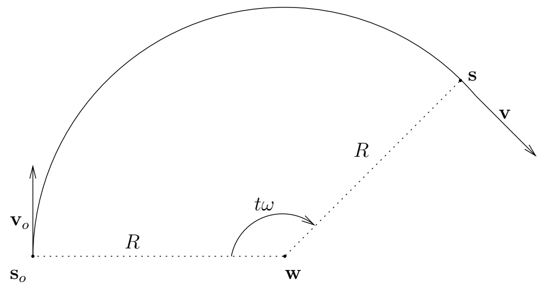

Fig. 1: Illustration of Theorem 1

3.2 Turn Dynamics in Java

The structural differences between PVS and Java play a large role in the way that the Java versions of algorithms are implemented. For example, some practices that are fairly common programming style in Java, such as exiting a program without returning a value, or returning a default failure value, are not possible in PVS. Another difference is that PVS functions must be provided all of their parameters, while a normal Java program may invoke or alter the values of any number of globally specified variables. For the kinematics library, all of the algorithms are written in Java as static methods to better reflect the functional specification style in PVS. Figure 2 illustrates the implementation of the function `turnOmega`, specified by Formula (1), in Java. While differences are apparent, the two versions are closely matched.

4 Model Animation of ATM Kinematic Library

The specification of an algorithm in a theorem prover such as PVS, along with an appropriate collection of theorems showing that the algorithm produces the desired output, allows for an extraordinarily high level of assurance that the algorithm is designed and implemented correctly. The translation of such an algorithm in a syntactically close way from the formal models to code carries along

```
static Pair<Vect3,Velocity> turnOmega(Vect3 s0, Velocity v0, double t,
  double omega) {
  if (Util.almost_equals(omega,0))
    return new Pair<Vect3,Velocity>(s0.linear(v0,t),v0);
  double v = v0.gs()/omega;
  double theta = v0.trk();
  double xT = s0.x + v*(Math.cos(theta) - Math.cos(omega*t+theta));
  double yT = s0.y - v*(Math.sin(theta) - Math.sin(omega*t+theta));
  double zT = s0.z + v0.z*t;
  Vect3 ns = new Vect3(xT,yT,zT);
  Velocity nv = v0.mkTrk(v0.trk()+omega*t);
  return new Pair<Vect3,Velocity>(ns,nv);
}
```

Fig. 2: Java implementation of `turnOmega`

much of this assurance of correctness. The final step in the proposed validation process is to evaluate the formal models on a selected collection of test inputs and to compare the outputs of this evaluation to the outputs computed by the code.

The reason why this model animation is important is two-fold. First, the algorithms that are examined invariably rely on simpler functions, which in turn rely on simpler functions, and so on, down to the basic functions defined in each respective language. While the syntactic similarity of the formal models and their implementations suggests that they perform the same computation, even slight differences in the lower-order functions could introduce significant differences in behavior. The comparison of the outputs of the two versions of the algorithm on a wide range of inputs can catch these invisible differences.

The second reason for the comparison of Java and PVS outputs is due to the inherent differences between how PVS and Java operate on numerical values. In PVS, like almost every other specification language, numerical operations are defined over real numbers. In contrast, Java, like almost every other programming language, uses floating point arithmetic. This means that for any algorithm which manipulates numerical values, calculations in Java may introduce estimations that make the output slightly different than what the calculation would produce if performed over the real numbers. For any one calculation, the difference between the expected real number output and the floating point estimate are generally small (on the order of 10^{-15}), but for an algorithm that performs hundreds of calculations, the effect of compounding small errors may lead to noticeable differences.

4.1 Test Generation

As mentioned in Section 2, three methods were chosen for selecting the input data for testing the output of the PVS specified algorithms versus the Java counterparts. All three methods assume that an appropriate range has been chosen for each input variable of the function under consideration. If d is the number of input variables, the allowable range of input values forms a d-dimensional hyper-rectangle in $\mathbb{R}^d$. For concision, any such rectangle will be referred to simply as a *box*.

The first method, which will be referred to as the *random* method, chooses a user specified number of points uniformly at random from the defined box. This method benefits from being simple to implement, and continuable in the sense that additional testing is unlikely to duplicate points, so further tests can be easily combined with previous results. The randomness aspect also helps mitigate the possibility that the outputs of a function match well for whole numbers or simple fractions, but not for long decimal expansions.

In the second method, referred to as the *grid* method, the user specifies a mesh size for each variable in the function being tested. For example, suppose the range for the variable t is $0 \leq t \leq 2$, and a mesh size of 0.5 is specified. Then the range for the variable t is split into the 4 subintervals $[0, 0.5]$, $[0.5, 1]$, $[1, 1.5]$, and $[1.5, 2]$. In general, if the range of variable t is $[a, b]$, and the mesh size is ϵ_t, then the number of subintervals created is roughly $(b - a)/\epsilon_t$. As should be expected, if the variable range is large, or the mesh size is small, the number of subintervals created can be very large. Each endpoint of a subinterval is used as a possible input for the given variable. For example, there are 5 inputs for the variable t above. These values are calculated for each variable, and every combination of the values is used as a test point. Essentially, the original range box is sliced in each dimension, and the intersection points of the slices are taken as test points for the function. The major benefit of this method versus the random method is guaranteed coverage of the input space. The main drawbacks are that the number of points created can be very large (depending on the number of variables, their ranges, and mesh sizes), and that the points tested may not represent "average" points, since they lack the randomness element of the first method.

The third method, called the *grid random* method, combines these two techniques. It first splits the range of each variable into subintervals using a user defined mesh size. Note that every possible choice of one such subinterval for each variable defines a sub box of the original range. This method selects one point uniformly at random from within each such box. This method benefits from the guaranteed coverage of the grid method, and the randomness of the random method, but is the most computationally expensive of the three.

The combination of the three methods above offer a number of advantages, in that they are simple to describe and implement while also allowing any plausible input value a non-zero probability of being selected. The grid and grid-random methods also provide for fairly uniform coverage of the input space. On the other hand, the methods above do not necessarily satisfy any code coverage criterion, such as MC/DC [12]. In general, *any* method that can generate test cases from either the software implementation or the formal specification can be used to produce the test inputs. Some of these possibilities are discussed in Section 5.

Once a method for determining function inputs has been selected, the following four steps must be performed in order to compare the outputs of a function under test.

1. Generate the set of test points according to the specified testing method.
2. Determine the output of the Java version of the function on each test point.
3. Determine the output of the PVS version of the function on each test point.
4. Compare the values of the two outputs, to determine if they agree up to some user-defined tolerance.

Because the Java versions of each function are purposely built for computation, and the actual inputs from end-users will be processed through Java, the first two steps are carried out using Java. To do this, a Java program was written specifically for each function and testing method. The output of the Java program is a collection of text files, each containing a list of formatted records in PVS syntax. Each record consists of a single test point, which lists the floating point input value of each variable in the function being tested, as well as the floating point output of the Java version of the function on evaluation at the test point.

4.2 Model Animation

To determine the output of a PVS function on a particular input, it is necessary to be able to evaluate the function on concrete input values. In PVS, this can be done through the ground evaluator [24] assuming that the functions are written in the executable fragment of PVS. Most functions in the ATM kinematic library are a priori computable, except that they rely on non-computable real-number functions such as square root and trigonometric functions. The PVS ground evaluator does not support evaluation of these kinds of functions.

To evaluate functions that are not supported by the ground evaluator, it is necessary to use semantic attachments [9]. A semantic attachment is a piece of code that links an uninterpreted PVS function to another function, possible another PVS function, for the sake of evaluation. For example, the square root function cannot be exactly evaluated, since it often returns an irrational number on a rational input. In a formal specification on the other hand, the precise square root function can be reasoned about, and properties proven about it theoretically. If a function in this formal specification is to be evaluated, some computable method to approximate the square root, the semantic attachment, is provided. Any time a square root is encountered in the execution of the specification, the semantic attachment is evaluted instead.

In general, semantic attachments are not safe as there is no guarantee that the semantic attachments soundly and completely realize the original functions. For instance, it is impossible to provide safe semantic attachments to irrational real-valued functions. Indeed, a semantic attachment has no guarantee to have any relation to the function it is attached to. Hence, in PVS, semantic attachments are allowed in the animation of specifications, but not in a formal proof.

Since writing semantic attachments is error prone, PVS includes the animation tool PVSio [19] that provides a predefined library of semantic attachments. The PVSio library of semantic attachments includes input/output operations, imperative features, and floating point arithmetic. For this project, PVSio has been extended with semantic attachments for exact arithmetic definitions of square root, sine, cosine, and arctangent. Concretely, if f is one of these mathematical functions, a semantic attachment `f_sa` is provided that satisfies the following property for all $x \in \mathbb{R}$

$$|\texttt{f\_sa}(x) - f(x)| \leq \epsilon, \tag{2}$$

where ϵ is a small positive number provided by the user. With these semantic attachments, all evaluations are then performed using exact arithmetic. However, it should be noted that Formula (2) does not guarantee that the computational error is always bounded by ϵ, as errors accumulate when combined in large numerical expressions. Overall, these semantic attachments provide a much better numerical precision than floating-point arithmetic and, since arithmetic is always exact for all the other operators, evaluation of numerical expressions is independent of the order of evaluation.

For this project, PVSio has also been extended with a library of semantic attachments that automate the process of checking test files in the format discussed in Section 4.1. This library provides functionality for reading text files, converting floating point inputs into exact rational number representations, symbolically evaluating these rational inputs in PVS, comparing the outputs to a given tolerance, and printing the results. This library, which is called PVSioChecker, is now part of the NASA PVS libraries.

4.3 Results

The five functions that were chosen for comparison between the PVS and Java versions were the following:

From the vertical speed algorithms, the functions tested were

- `vsAccelUntil`,
- `vsAccelUntilWithRampUp`,
- `vsLevelOut`.

From the ground speed algorithms, the function tested was

- `gsAccelUntil`,

From the turn algorithms, the function tested was

- `turnOmega`.

Each function was tested using all three test-point selection methods (random, grid, and grid random), where the upper and lower bounds for the majority of the parameters come from Stratway defaults. The only parameters lacking default values in Stratway are the horizontal position coordinates. For these, upper and lower bounds were chosen to be 1000 and -1000 nautical miles in each coordinate. Several of the bounds apply to multiple parameters. For instance, the bounds on ground speed apply to both the initial ground speed of the aircraft, and to the goal ground speed used in a `gsAccelUntil` maneuver. The parameters and corresponding bounds are listed in Table 1.

Table 1: Global bounds for input parameters.

	$\mathbf{s}_{ox}, \mathbf{s}_{oy}$	altitude	ground speed	track angle
lower	-1000 nmi	500 ft	50 kn	0 deg
upper	1000 nmi	40,000 ft	700 kn	360 deg

	vertical speed	bank angle	acceleration[1]
lower	-5000 ft/min	-30 deg	0.1 m/s^2
upper	5000 ft/min	30 deg	2 m/s^2

[1]Bounds apply to ground speed and vertical speed acceleration.

The output of each function on a test point is a pair of vectors containing a calculated position and velocity for some point of the chosen maneuver. Given a test point, this pair of vectors is computed using both the PVS and the Java versions of the function, and if the PVS and Java outputs for any single coordinate differ by more than a tolerance value, which is set to 10^{-8}, the test point is marked as a *fail*. The precision used for the semantic attachments of real-valued functions is 10^{-15}. In general, the threshold for tolerance will depend on the particulars of the software under consideration. For the software considered here, the input data are positions and velocities of aircraft, which in the use-case are obtained through the Automatic Dependent Surveillance - Broadcast (ADS-B) system on each aircraft. At the highest level of fidelity that these systems may be certified at, the horizontal position is required to be accurate to with 3 meters, the vertical position accurate to within 45 meters, and the horizontal velocity accurate to within 0.3 meters/second[5] [1]. The minimum acceptable standards are far less precise. All calculations are performed in these units, and so a tolerance of 10^{-2} would likely be sufficient in this case. The tolerance used, 10^{-8} was selected because it reveals the edge of where the Java and PVS implementations differ. The precision for the semantic attachments was chosen through trial and error to be as small as possible without significantly increasing the computation time.

[5] There is no requirement for vertical velocity accuracy.

For each function and point selection method, Table 2 lists the number of records created, the number of fails, and the CPU time of testing.[6] Approximately 2 million test points were generated for each function, spread fairly evenly over the three testing methods. For the random method, the number of points to be tested is simple to explicitly specify. For the grid and grid random methods, the number of test points is governed by the step size chosen for each parameter. Each function has, as input, an initial 3D position and velocity, a time parameter, and some number of other parameters. Because each function has a parameter space of at least 8 dimensions, a decrease in step size by half in each parameter would result in at least 256 times as many records than before the decrease. Due to this, certain parameters of each function were given priority for allowing small step size. For instance, for the vertical speed algorithms, the altitude, vertical velocity, and vertical acceleration parameters were prioritized, since the horizontal position and velocity are simply projections in these algorithms. The step sizes were then calculated that would produce the desired number of test points.

In all, over 8 million test records were generated, and fewer than 0.01 % of the records failed with the specified tolerance of 10^{-8}. A few further notes about the results are in order. First, if the tolerance is increased to 10^{-6}, there are no failures at all. Second, the function testing whether two numbers are almost equal compares them in terms of *absolute* error. If compared in terms of *relative* error at the same tolerance, then there are again zero failures. Lastly, it is notable that almost all of the failures occurred in the function `turnOmega`. This is likely due to the function modeling a circular turn, while the other functions maintain straight-line trajectories. Because of this, the output of `turnOmega` is highly sensitive to any error in the calculation of several trigonometric functions. Nevertheless, a closer examination of the actual failures records for `turnOmega` was conducted. The examination revealed that nearly all failures occurred when the angular velocity parameter is below 0.2 deg/sec, and the time parameter is over 1000 seconds. This corresponds calculating a point over 16 minutes into a turn with a very slight bank angle. Such turns are rarely executed in reality, where the standard turn rate is approximately 3 deg/sec, taking just 2 minutes for a full 360 degree turn.

5 Related and Future Work

Model animation is a key feature of model-based development tools. For instance, MathWork's Simulink[7] is a widely-used simulation environment for the analysis of dynamical systems, which are specified using state charts. In the context of formal methods, tools like PVSio-web [16], which is also built on top of PVSio, and PetShop [22], which animate Petri nets, provide powerful features for prototyping and validating formal specifications. In [2], VDM models are animated

[6] All testing was performed on a 2014 Macbook Pro with a 2GHz Intel Core i7 processor and 8 GB of RAM.

[7] `http://www.mathworks.com/products/simulink`.

Table 2: Testing Results

`vsAccelUntil`	Records	Fails	CPU time
Rand	1,000,000	0	11.32 hr
Grid	622 ,080	0	4.11 hr
G-R	332,659	0	2.88 hr
totals	1,954,739	0	18.31 hr

`vsAccelUntilWithRampUp`	Records	Fails	CPU time
Rand	960,000	0	11.7 hr
Grid	340,416	0	2.45 hr
G-R	665,429	0	6.48 hr
totals	1,965,845	0	20.63 hr

`vsLevelOut`	Records	Fails	CPU time
Rand	810,000	0	11.53 hr
Grid	518,400	0	4.88 hr
G-R	915,000	8	11.42 hr
totals	2,243,400	8	27.83 hr

`gsAccelUntil`	Records	Fails	CPU time
Rand	330,000	0	12.29 hr
Grid	315,000	0	11.8 hr
G-R	340,000	0	11.7 hr
totals	985,000	0	35.79 hr

`turnOmega`	Records	Fails	CPU time
Rand	615,000	225	13.06 hr
Grid	504,000	300	7.89 hr
G-R	436,066	309	8.4 hr
totals	1,555,066	834	29.35 hr

Global Totals	Records	Fails	CPU time
Rand	3,715,000	225	59.9 hr
Grid	2,299,896	300	31.13 hr
G-R	2,689,154	317	40.88 hr
totals	8,704,050	842	131.91 hr

and used as oracles on generated test cases to uncover requirement errors. These works, however, do not aim at validating formal models against their software implementations as the approach proposed in this paper.

The approach presented in this paper is similar to the one supported by tools like QuickCheck [8] for Haskell and AutoTest [18] for Eiffel. These tools check software annotations on a set of randomly generated test cases. Similar tools exist for theorem provers [3] and other formal methods [26]. The presented approach also has similarities to the animation of EventB/B models using tools such as JeB [25] and ProB [14]. Indeed, JeB even provides support for a type of semantic attachment in the form of "hooks" for the user to supply Java code where a function in the specification is undefined. These tools, though, are generally intended for early testing of a specification, and for model checking. To the best knowledge of the authors, none of these tools attempt to bridge the gap between code and formal specifications due, for example, to numerical computations.

Concolic test [23] and other test generation techniques [7] combine concrete and symbolic execution of *code* to generate test cases that satisfy some coverage criteria. Generation of test cases is a step of the proposed approach. Hence, the software validation approach proposed in this paper can directly use these techniques. Indeed, an early reviewer of this paper suggested the following technique. Generate a test suite by determining a set of inputs that provide guaranteed path coverage on the formal specification, and another set of inputs that guarantee coverage on the software implementation. Using the full test suite would guarantee similar behavior of the software and its specification on every possible execution path for a concrete test value.

Future work involves employing the approach to validate more of the code utilized by the NASA air traffic management software under development, as well as further employing and developing tools to automate the code generation from specification. Another line of research is to develop a method for producing guaranteed output precision, or upper and lower bounds, for the symbolic evaluation of a function in PVS.

The NASA PVS library also contains a specification of floating point numbers and operations on them. Algorithms specified in this context can be translated to code in a more faithful way, and the behavior is likely to be much closer between the two. The hurdle to this pursuing this line of research is that proving properties of functions inside the context of floating point numbers is much more difficult.

6 Conclusion

Despite recent progress on the formal analysis of floating point programs [4,5,15], verification of software involving numerical computations is still a challenging problem. An alternative approach to software verification consists on the development of code from formally verified models of safety-critical algorithms. While this approach does not provide strong guarantees of software correctness, visual inspection of both the code and the formal models increases the confidence that the software behavior closely reflects its formal specification. This paper proposes a new approach that automates the validation of software implementations against their formal models. This approach, which is based on model animation, compares the output of algorithms implemented in a programming language to the results obtained from the symbolic evaluation of formal models enriched with semantic attachments. These semantic attachments enable symbolic evaluation of even irrational, real-valued functions, via precise numerical computations. The proposed approach is illustrated on an air traffic management system currently used at NASA for conducting research on advanced air traffic management concepts.

References

1. Federal Aviation Administration. Airworthiness approval of automatic dependent surveillance-broadcast (ads-b) out systems. Advisory Circular AC 20-165A, FAA,

Nov 2012.

2. Bernhard K Aichernig, Andreas Gerstinger, and Robert Aster. Formal specification techniques as a catalyst in validation. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 203–206. IEEE, 2000.
3. Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.
4. Sylvie Boldo. *Deductive Formal Verification: How To Make Your Floating-Point Programs Behave*. Thèse d'habilitation, Université Paris-Sud, October 2014.
5. Sylvie Boldo and Claude Marché. Formal Verification of Numerical Programs: from C Annotated Programs to Mechanical Proofs. *Mathematics in Computer Science*, 5:377–393, 2011.
6. Ricky Butler. Formalization of the integral calculus in the PVS theorem prover. *Journal of Formalized Reasoning*, 2(1), 2009.
7. Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM.
8. Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.
9. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001.
10. George Hagen, Ricky Butler, and Jeffrey Maddalon. Stratway: A modular approach to strategic conflict resolution. In *Preceedings of 11th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, Virgina Beach, VA, September 2011.
11. George E. Hagen and Ricky W Butler. Towards a formal semantics of flight plans and trajectories. Technical Memorandum NASA/TM-2014-218862, NASA, Langley Research Center, Hampton VA 23681-2199, USA, Dec 2014.
12. Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Memorandum NASA/TM-2001-210876, NASA, Langley Research Center, Hampton VA 23681-2199, USA, May 2001.
13. Leonard Lensink, Sjaak Smetsers, and Marko van Eekelen. Generating verifiable Java code from verified PVS specifications. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 310–325. Springer Berlin Heidelberg, 2012.
14. Michael Leuschel and Michael Butler. Prob: A model checker for b. In *FME 2003: FORMAL METHODS, LNCS 2805*, pages 855–874. Springer-Verlag, 2003.
15. Claude Marché. Verification of the functional behavior of a floating-point program: an industrial case study. *Science of Computer Programming*, 96(3):279–296, March 2014.
16. Paolo Masci, Patrick Oladimeji, Paul Curzon, and Harold Thimbleby. Tool demo: Using PVSio-web to demonstrate software issues in medical user interfaces. In *4th International Symposium on Foundations of Healthcare Information Engineering and Systems (FHIES2014)*, 2014.

17. Bertrand Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, October 1992.
18. Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42(9):46–55, Sept 2009.
19. César Muñoz. Rapid prototyping in PVS. Contractor Report NASA/CR-2003-212418, NASA, Langley Research Center, Hampton VA 23681-2199, USA, May 2003.
20. Anthony Narkawicz and César Muñoz. State-based implicit coordination and applications. Technical Publication NASA/TP-2011-217067, NASA, Langley Research Center, Hampton VA 23681-2199, USA, March 2011.
21. Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceeding of the 11th International Conference on Automated Deductioncade*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, June 1992.
22. Philippe Palanque, Jean-François Ladry, David Navarre, and Eric Barboni. High-fidelity prototyping of interactive systems can be formal too. In Julie A. Jacko, editor, *Human-Computer Interaction. New Trends*, volume 5610 of *Lecture Notes in Computer Science*, pages 667–676. Springer Berlin Heidelberg, 2009.
23. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.
24. Natarajan Shankar. Efficiently executing PVS. Technical report, Project report, ComputerScience Laboratory, SRI International, Menlo Park, 1999.
25. Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières. Jeb: Safe simulation of event-b models in javascript. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, volume 1, pages 571–576, Dec 2013.
26. Wada Yusuke and Kusakabe Shigeru. Performance evaluation of a testing framework using quickcheck and hadoop. *IPSJ Journal*, 53(2):7p, feb 2012.