Kalpa Publications in Computing

Volume 5, 2018, Pages 93–108

Automated Formal Methods

The MINERVA Software Development Process

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

NASA Langley Research Center
MS 130
Hampton, Virginia, U.S.A.

Abstract

This paper presents a software development process for safety-critical software components of cyber-physical systems. The process is called MINERVA, which stands for Mirrored Implementation Numerically Evaluated against Rigorously Verified Algorithms. The process relies on formal methods for rigorously validating code against its requirements. The software development process uses: (1) a formal specification language for describing the algorithms and their functional requirements, (2) an interactive theorem prover for formally verifying the correctness of the algorithms, (3) test cases that stress the code, and (4) numerical evaluation on these test cases of both the algorithm specifications and their implementations in code. The MINERVA process is illustrated in this paper with an application to geo-containment algorithms for unmanned aircraft systems. These algorithms ensure that the position of an aircraft never leaves a predetermined polygon region and provide recovery maneuvers when the region is inadvertently exited.

1 Introduction

The formal verification of safety-critical software for cyber-physical systems presents multiple challenges. Since these systems interact with the environment, their functional and operational requirements may involve complicated properties that are beyond the reach of automated analysis techniques. Formal verification of cyber-physical systems is also challenging because machine numbers such as floating-point numbers are used to implement numerical computations. Finally, although embedded systems avoid imperative features such as dynamic memory and unbounded loops, their control flow often relies on numerous conditional statements that can generate an enormous number of potential execution paths.

This paper presents a practical, but rigorous, approach to the development of safety-critical software components of systems that interact with the environment. The process is called MINERVA, which stands for Mirrored Implementation Numerically Evaluated against Rigorously Verified Algorithms, and has formal methods as its centerpiece. In MINERVA, functional and operational requirements are first specified using a formal specification language. Core algorithms that implement those requirements are also specified and formally proved correct with respect to their specifications. These algorithms are then numerically evaluated on a generated set of test cases. Finally, the output values are compared to outputs computed by an implementation of these algorithms in a programming language with the purpose of showing similar behavior between the algorithm specifications and their corresponding implementations in code.

MINERVA has been used in the development of numerous software prototype implementations of NASA's air traffic management concepts, e.g., DAIDALUS[1] [12], ICAROUS[2] [6], and PolyCARP [3] [13]. This paper illustrates the MINERVA process through PolyCARP, a suite of algorithms for computations on polygons that is used in geo-containment applications. In particular, PolyCARP is used inside the ICAROUS software package to provide geo-containment and obstacle-avoidance capabilities to small unmanned aircraft. The geo-containment functionality of the PolyCARP package uses polygon containment algorithms to determine whether the position of an aircraft is within a given geographical region, which is modeled using a 2D polygon with a minimum and a maximum altitude. For safety critical aircraft systems such as geo-containment systems, formal verification and validation is key to having assurance of the safe behavior of the software, making such systems good targets for the application of MINERVA. To illustrate the practical benefits of the MINERVA process, four software bugs are shown that were found and fixed using MINERVA on the algorithms in PolyCARP.

MINERVA does not assume a particular specification language, proof assistant, or programming language. The specification language should be expressive enough to support the specifications of the continuous behavior of the environment, the control logic of the algorithms, and the correctness properties of these algorithms. The proof assistant should be able to support the formal verification of these algorithms and should provide a ground evaluator of numerical properties. The examples of the MINERVA process presented in this paper use SRI's Prototype Verification System (PVS) [16] as a specification language and proof assistant, and the programming languages Java, C++, and Python. The PVS specifications are viewed as the formal definitions of both the algorithms and safety properties.

The rest of this paper is organized as follows. Section 2 describes the MINERVA development process. Section 3 presents the application of MINERVA to PolyCARP. Sections 4 and 5 discusses related work and conclude this paper.

2 MINERVA

The MINERVA software development process, short for Mirrored Implementation Numerically Evaluated against Rigorously Verified Algorithms, is defined by three main steps, which correspond the three main parts of this acronym (although not in order):

Rigorously Verified Algorithms. Formal specification of algorithms, formal specification of their requirements, and formal proofs of their correctness in a proof assistant.

Mirrored Implementations. Manual or automatic implementations of the verified algorithms in code.

Numerically Evaluated. Using a set of test inputs, calculate the output values of both the formal algorithm specifications and their software versions and compare them to ensure faithful translation of the algorithm specifications to code. The process in this step is called model animation [9].

It can be seen from this description that the MINERVA process provides high assurance that the algorithms are both mathematically correct and faithfully translated into code. Figure 1

[1]http://shemesh.larc.nasa.gov/fm/DAIDALUS.
[2]http://shemesh.larc.nasa.gov/fm/ICAROUS.
[3]http://shemesh.larc.nasa.gov/fm/PolyCARP.

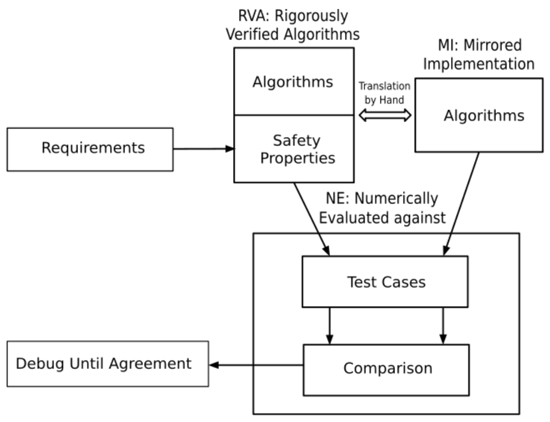

Figure 1: MINERVA Development Process

illustrates the entire MINERVA process in diagrammatic form. The following sections describe each of the steps in MINERVA in greater detail.

2.1 Rigorously Verified Algorithms

The Rigorously Verified Algorithms (RVA) step in the MINERVA process can be broken into three substeps:

1. Formal specification of algorithms.
2. Formal specification of algorithm requirements, e.g., correctness or safety properties.
3. Formal proofs that algorithms satisfy requirements and safety properties.

The examples in this paper accomplish each of these three steps using the PVS theorem prover. The RVA step in the MINERVA process is illustrated for geo-containment algorithms in Section 3.1.

2.2 Mirrored Implementation

The Mirrored Implementation (MI) step in the MINERVA process refers to the algorithm specifications from the RVA step being translated to code in a programming language. There are two main ways that this can be accomplished:

- Translation of the algorithms by hand between the specification language and the programming language
- Automatic-generation of code in the programming language from the algorithm specifications, possibly using a tools such as the one described in [10].

The exact method of translation between specifications and code is not prescribed by the MINERVA proces. Either hand translation of automatic generation could potentially be unsound. However, it should be noted that the use of an automatic generation tool may further improve the reliability of code written using the MINERVA process, especially if the translation logic is formally verified itself.

An important aspect of the MI step is that the interfaces to the algorithm specifications and their implementations are identical, hence the word *mirrored*. This allows comparison of outputs of the individual functions on a suite of test cases in the next step.

2.3 Numerically Evaluated

The Numerically Evaluated (NE) step in the MINERVA process involves two substeps:

1. Test case generation. In this substep, a suite of test cases is generated from the formal models for stressing the algorithms and their implementations.
2. Agreement testing. Using animations of the algorithm specifications, compute outputs for both the algorithms and their software implementations on the generated suite of test cases. Finally, compare the outputs by using a conformance relation between corresponding values.

These steps have been presented before as the concept of *model animation* [9]. While model animation does not provide an absolute guarantee that software implementations are correct, it increases the confidence that the formal models are faithfully implemented in code.

Despite this structural similarity between the algorithms and the code, the execution of the functional models and the software implementations may differ due to the presence of functions that cannot be effectively computed, such as trigonmetric functions. In this case, semantic attachments [7] can be supplied for such atomic functions to make these functions executable in the specification language. The NASA PVS Library[4] includes several formalizations of rigorous numerical approximation methods and a computable high-level formalization of floating-point number that can be used as semantic attachements for real arithmetic operators.

The generation of test cases in this step is critical to its success. In practice, any method of generating these tests can be used, including both user-provided test cases and automatically generated test cases. For the geo-containment algorithms presented in Section 3, these test cases are generated by the developer in a way designed to stress the control logic in the algorithms.

3 Application of MINERVA to PolyCARP

This section illustrates the MINERVA process through its application to the PolyCARP package, which provides algorithms for computations on polygons, with geo-containment of unmanned systems being one intended application. As noted in the Introduction, the specification and prover languages of PVS are used for specifying and verifying these polygon algorithms. The final code, produced in the MI (Mirrored Implementation) step of MINERVA, is implemented in Java, C++, and Python. The geo-containment functionality of the PolyCARP package uses standard polygon containment algorithms. The algorithms test that a position is inside a given polygon region and provide nearby resolution locations to recover to in the event that an unsafe region is breached. For instance, a resolution function is given with a point and a polygon as inputs that, when the point is inside the polygon, returns another point that is close to the first point but is outside the polygon. To illustrate the practical benefits of the process, four software bugs are shown in Section 3.3 that were found and fixed using MINERVA on the algorithms in PolyCARP.

[4] `https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/`.

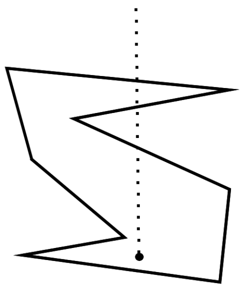
Figure 2: Ray Casting

3.1 Rigorously Verified Algorithms

This section illustrates the RVA step in the MINERVA process through the geo-containment algorithms in the PolyCARP package. These algorithms perform computations on polygons.

3.1.1 Formal Specification of Algorithms for Polygons

The containment functions for 2D polygon regions assume that any input polygon is arranged in *counterclockwise* order. These functions are based, in part, on *ray casting.* Given a polygon region and an input position in a 2D plane, a ray is cast from the point outward to infinity (in this case the direction of the positive y-axis). In most cases, if the ray crosses an even number of edges of the polygon, it is outside; otherwise, it is inside. This is shown in Figure 2.

In PVS, the function `ray_crosses`, defined below, determines whether this ray crosses the i-th edge of the polygon $\boldsymbol{P}$, where `s` refers to the position to be tested for containment in $\boldsymbol{P}$.

```
ray_crosses($P$,s,$i$) $\equiv$
  let
    N=size(P),
    next = mod(i+1,N),
    tester = $(p(next)_x - p(i)_x)$^2 $\cdot$ $(p(i)_y - s_y)$ + $(s_x-$
      $p(i)_x)$ $\cdot$ $(p(next)_y - p(i)_y)$ $\cdot$ $(p(next)_x - p(i)_x)$
  in
    if $p(i)_x$>$s_x$ and $p(next)_x$>$s_x$ then false
    elsif $p(i)_x$<$s_x$ and $p(next)_x$<$s_x$ then false
    elsif $p(i)_x$ = $p(next)_x$ and ($p(i)_y \geq s_y$ or
        $p(next)_y \geq s_y$) then ERROR
    elsif $p(i)_x$ = $p(next)_x$ then false
    elsif tester $\geq$0 then true
    else false
    fi
```

In the containment method for 2D polygons, a *buffer* distance named `BUFF` is also used to perturb the original polygon $\boldsymbol{P}$ because ray casting along the direction of the y axis can sometimes cause the ray to pass very close to some vertices, which can potentially allow floating point errors to produce an incorrect inside/outside result. The perturbation of the vertices of $\boldsymbol{P}$ by `BUFF` stops this from happening, and the ray casting function then works as expected on the perturbed polygon $\boldsymbol{P}^*$. In PVS, there are functions `definitely_in`($\boldsymbol{P}$, `s`, `BUFF`) and `definitely_out`($\boldsymbol{P}$, `s`, `BUFF`) that take as inputs the polygon $\boldsymbol{P}$, the position `s`, and the buffer distance `BUFF` and return a boolean value. These functions are based partly on ray casting (and also winding numbers) and are the basic containment functions used in PolyCARP. Their formal definitions can be found in the PVS development. In addition to these standard containment methods, the function `definitely_in` (respectively, `definitely_out`) checks that the first edge crossed by the ray that is cast crosses it from right to left (respectively, left to right). Thus, for either of these functions to return *true*, the polygon must be in counter-clockwise order. The function `definitely_out` is therefore used to check that the polygon is in counter-clockwise order by inputting a point that is known to be outside the polygon and checking that this function return *true*.

One way that the counter-clockwise property is checked is by the computation of two *extremal* vertices of the polygon and checks that the edges of the polygon make a *left* turn at these vertices. These extremal points are computed using the recursive (on the vertex index i) function `counterclockwise_corner_index`.

```
counterclockwise_corner_index($P, \epsilon, i$) $\equiv$
    if i=0 then i
    else
      let j = counterclockwise_corner_index($P, \epsilon, i-1$) in
      if $p(j)_x$=$p(i)_x$ and $p(j)_y \geq p(i)_y$ then i
      elsif $p(j)_x$=$p(i)_x$ then j
      elsif $\epsilon \cdot p(j)_x > \epsilon \cdot p(i)_x$ then i
      else j
      fi
    fi
```

If the recursive input i is set to `size`($\boldsymbol{P}$)-1, then this function covers every possible vertex index, and setting $\epsilon = 1$ returns one extremal vertex index, and $\epsilon = -1$ returns another. The function `cc_edges` checks that $\boldsymbol{P}$ turns right at index `counterclockwise_corner_index`($\boldsymbol{P}, -1,$ `size`($\boldsymbol{P}$)-1) and makes another right turn at index `counterclockwise_corner_index`($\boldsymbol{P}, 1,$ `size`($\boldsymbol{P}$)-1).

Many of the functions in the PVS development also depend on the polygon having the property that no two, non-adjacent edges come within the (small) distance `BUFF` of each other. To compute this, a function `quad_min_box` is called that determines whether a bivariate quadratic $ax^2+by^2+cxy+dx+ey+f$, with $a \geq 0$ and $b \geq 0$, ever falls below a value D when $x \in [0,1]$ and $y \in [0,1]$. The function `segments_2D_close`, defined below, uses the function `quad_min_box` to determine whether a line segment between the points s_1 and e_1 comes within distance `BUFF` of the line segment between the points s_2 and e_2.

```
segments_2D_close($s_1, e_1, s_2, e_2$, BUFF) $\equiv$
  let
    segXApart=$|(s_{1x} - s_{2x})| > 2 \cdot$ BUFF and
      $|(s_{1x} - e_{2x})| > 2 \cdot$ BUFF and
      $|(e_{1x} - e_{2x})| > 2 \cdot$ BUFF and
      $|(e_{1x} - s_{2x})| > 2 \cdot$ BUFF and
      $sign(s_{1x} - s_{2x}) = sign(s_{1x} - e_{2x})$ and
      $sign(e_{1x} - e_{2x}) = sign(e_{1x} - s_{2x})$ and
      $sign(s_{1x} - s_{2x}) = sign(e_{1x} - e_{2x})$,
    segYApart=$|(s_{1y} - s_{2y})| > 2 \cdot$ BUFF and
      $|(s_{1y} - e_{2y})| > 2 \cdot$ BUFF and
      $|(e_{1y} - e_{2y})| > 2 \cdot$ BUFF and
      $|(e_{1y} - s_{2y})| > 2 \cdot$ BUFF and
      $sign(s_{1y} - s_{2y}) = sign(s_{1y} - e_{2y})$ and
      $sign(e_{1y} - e_{2y}) = sign(e_{1y} - s_{2y})$ and
      $sign(s_{1y} - s_{2y}) = sign(e_{1y} - e_{2y})$
    in
      if segXApart or segYApart then false
      elsif near_edge($s_2, e_2, s_1$, BUFF) then true
      elsif near_edge($s_2, e_2, e_1$, BUFF) then true
      elsif near_edge($s_1, e_1, s_2$, BUFF) then true
      elsif near_edge($s_1, e_1, e_2$, BUFF) then true
      elsif $s_1 = e_1$ or $s_2 = e_2$ then false
      else
        let
          $s = s_1 - s_2$,
          $v = e_1 - s_1$,
          $w = e_2 - s_2$,
          $a = \|v\|$^2,
             $b = \|w\|$^2,
          $c = -2 \cdot (v \cdot w)$,
          $d = 2 \cdot (s \cdot v)$,
          $ee = -2 \cdot (s \cdot w)$,
          $f = \|s\|$^2
        in
          quad_min_box($a, b, c, d, ee, f, sq$(BUFF))
      fi
```

In order for containment and resolution algorithms to work well in practice, their input polygons are checked for several properties.

- The vertices are in counterclockwise order.
- No two non-adjacent edges come within the small distance BUFF of each other.
- No two adjacent edges meet at a sharp angle.

In the formal PVS development, the function nice_poly_2D, which is defined below, checks these properties. It uses a function called test_point_below, which returns a point with x coordinate between the minimum and maximum x coordinates of $\boldsymbol{P}$ and y coordinate below the minimum y coordinate of $\boldsymbol{P}$.

```
nice_poly_2D($P$, BUFF) $\equiv$
  let $N$ = size($P$) in
  cc_edges($P$) and
  definitely_out($P$, test_point_below($P$, BUFF), BUFF) and
  for all i=0,...,N-1: for all j=i,...,N-1:
    let
      mj = mod(j+1,N),
      mi = mod(1+i,N)
    in
      if i=j true
```

```
elsif p(i)=p(j) then false
elsif j=mi and near_edge($N, p, p(mj)$, BUFF, $i$) or
  near_edge($N, p, p(i)$, BUFF, $j$)) then false
elsif j=mi and corner_lt3_deg($p(j) - p(i), p(mj) - p(j)$)
  then false
elsif j=mi then true
elsif i=mj and (near_edge($N, p, p(mi)$, BUFF, $j$) or
  near_edge($N, p, p(j)$, BUFF, $i$))
  then false
elsif i=mj and corner_lt3_deg($p(i) - p(j), p(mi) - p(i)$)
  then false
elsif i=mj then true
elsif segments_2D_close($p(i), p(mi), p(j), p(mj)$, BUFF)
  then false
else true
fi
```

Once an unsafe region, either the inside or the outside of a particular polygon, is breached, *resolution algorithms* are provided that suggest a new position to maneuver to in order to exit the region. In addition to the parameter BUFF used in the algorithms above, the resolution algorithms have a distance parameter ResolBUFF. The algorithms suggest resolution points that are least ResolBUFF away from the polygon boundary. In many cases, the algorithms simply find the closest point on the boundary and suggest a point ResolBUFF away from this point, perpendicular to this edge. In the event that this suggested point is not definitely inside (respectively, outside), such as in some cases when it is near a vertex $p(i)$, another position is chosen a certain distance from $p(i)$ along the vector proj_vect$(p(i-1), p(i), p(i+1))$, where the function proj_vect is defined as follows.

```
proj_vect($u, v, w$) $\equiv$
  if $(v-u)\cdot\ (w-v)\ \geq\ 0$ then $\widehat{(v-u)}^{\perp} + \widehat{(w-v)}^{\perp}$
  elsif $(w-v)\cdot\ (v-u)^{\perp}\ \leq\ 0$ then $(v-u)^{\perp} + (v-w)^{\perp}$
  else $\hat{(u-v)} + \hat{(w-v)}$
  fi
```

As shown in Figure 3, the function proj_vect computes a direction vector to the *right* of the intersection of the line segments $u - v$ and $v - w$. In this definition, $a^{\perp}$ is defined by $(a_y, -a_x)$ and $\hat{a}$ is defined by $(1/\|a\|) \cdot a$ for any vector a. The function that finds the index of the closest edge to the input position s is defined in PVS using a recursive function (on the edge index i) according to the following formula.

```
closest_edge($P$, s, $i$) $\equiv$
  if i = 0 then i
  else
    let $ce$ = closest_edge($P$, s, $i-1$),
         $nexti$ = $mod(i+1, size(P))$,
         $closp$ = closest_point($p(i), p(nexti)$, s),
         $dist$ = $\|s - closp\|$,
         $nextce$ = $mod(ce+1, size(P))$,
         $prevclosp$ = closest_point($p(ce), p(nextce)$, s),
         $prevdist$ = $\|s - prevclosp\|$ in
```

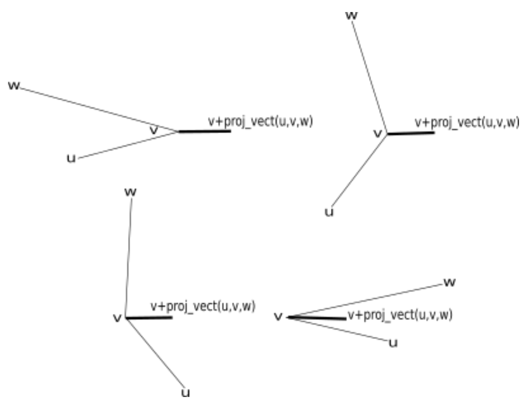

Figure 3: Direction vector computed by `proj_vect`

```
if $dist < prevdist$ then i
else $ce$
fi
```

The function `closest_edge` should be called with $i = \texttt{size}(\boldsymbol{P}) - 1$ to find the index of the closest edge of the entire polygon $\boldsymbol{P}$. While the definition of `closest_edge` seems somewhat trivial, it is included here because it will be mentioned later as an example of a function whose implementation in software had a bug that was found through the MINERVA process.

The PVS development defines two resolution functions:

$$\texttt{inside\_recovery\_point}(\boldsymbol{P}, \mathrm{s}, \texttt{BUFF}, \texttt{ResolBUFF}),$$
$$\texttt{outside\_recovery\_point}(\boldsymbol{P}, \mathrm{s}, \texttt{BUFF}, \texttt{ResolBUFF}).$$

These functions suggest points either inside or outside (respectively) to maneuver to in the event that an unsafe region is breached. Their definitions can be found in the PVS development and depend on both `proj_vect` and `closest_edge`.

3.1.2 Verification of Polygon Algorithms

The containment, well-formedness, and resolution algorithms all have certain properties proved about them in PVS. Several of these properties are stated below. The proofs of the following theorems use basic algebra, including some uncommon applications of the quadratic formula. Thus, they are mathematically accessible to most engineers. However, the algorithms have many conditional statements that complicate the control flow. Verifying the proofs in PVS is crucial to ensure that cases are correctly handled by the algorithms. In short, the algorithms and proofs are mathematically involved but not logically deep.

Theorem 1. *ray_crosses*$(\mathbf{P}, \mathbf{s}, i) =$ *true if and only if* $\exists r, t \in \mathbf{R} : r \geq 0,\ t \geq 0,\ t \leq 1,$ *and* $(1-t) \cdot p(i) + t \cdot p(mod(i+1, \textit{size}(\mathbf{P}))) = \mathbf{s} + (0, r)$.

Theorem 2. *Let* $cci =$ *counterclockwise_corner_index*$(\mathbf{P}, \epsilon, i)$ *where* $i <$ *size*$(\mathbf{P})$. *For every* $j \leq i$, $\epsilon \cdot p(j)_x \geq \epsilon \cdot p(cci)_x$ *and either* $p(j)_x \neq p(cci)_x$ *or* $p(j)_y \geq p(cci)_y$.

Theorem 3. *segments_2D_close*$(s_1, e_1, s_2, e_2, \textit{BUFF}) =$ *true if and only if* $\exists (w, v \in \mathbf{R}^2)$: $\|w - v\|^2 < \textit{BUFF}^2$ *and* *on_segment?*$(s1, e1, v)$ *and* *on_segment?*$(s2, e2, w)$.

Theorem 4. $\|\texttt{proj\_vect}(u,v,w)\| > 1$.

Theorem 5. *Suppose* $\texttt{nice\_poly\_2D}(\mathbf{P}, \mathit{BUFF})$ *holds. Set* $N = \texttt{size}(\mathbf{P})$ *and define* $ce = \texttt{closest\_edge}(\mathbf{P}, \mathbf{s}, N-1)$*. Choose any* $i < N$*. Let* $q_i = \texttt{closest\_point}(p(i), p(mod(i+1, N)), \mathbf{s})$ *and* $q_{ce} = \texttt{closest\_point}(p(ce), p(mod(ce+1, N)), \mathbf{s})$*. Then* $\|\mathbf{s} - q_i\| \geq \|\mathbf{s} - q_{ce}\|$*.*

Theorem 6. *If* $irp = \texttt{inside\_recovery\_point}(\mathbf{P}, \mathbf{s}, \mathit{BUFF}, \mathit{ResolBUFF})$*, then it follows that* $\texttt{definitely\_in}(\mathbf{P}, irp, \mathit{BUFF})$ *holds.*

3.2 Mirrored Implementation

The MI (Mirrored Implementation) step in the MINERVA process for the geo-containment algorithms in the PolyCARP package was mostly straighforward. The formal PVS specifications of the algorithms were translated to software as C++, Java, and Python code, which can be found in the PolyCARP repository. Recursive functions in PVS, for example the function `counterclockwise_corner_index`, were implemented in code using loops. This translation to code was done by hand, as in other applications of the MINERVA process to date [12, 6]. However, there is nothing that precludes automatic generation of code from PVS specifications in the future. If such a code generator was used, both the verification of the generator and the Numerically Evaluated step (also called model animation [9]) would contribute to correctness argument for the final code.

3.3 Numerical Evaluation

This subsection illustrates the NE (Numerical Evaluation) step in the MINERVA process through the geo-containment algorithms in the PolyCARP package. The Numerical Evaluation step of the MINERVA approach, also known as model animation [9] entails comparing symbolic output values of formal specifications to actual output values of implemented code to ensure faithful implementations. This subsection presents the results of the numerical evaluation process on the polygon algorithms in PolyCARP. Four bugs are presented that were found and fixed using the model animation process. These were bugs in both the C++ and Java versions of the resolution algorithms that suggest a new point inside/outside a polygon region to maneuver to once an undesirable or unsafe region has been breached.

3.3.1 Test Case Generation

The first step of model animation is to generate a large number of input values for the algorithms. These will be used to compare the result of evaluation of these values in the formal models to those computed by the software implementation. The goal is to find input values that are likely to stress the algorithms and find potential differences between the implementations. For the PolyCARP algorithms, the test cases consist of the following:

- A large number of random polygons (with units in meters).
- A large number of test points for each polygon.

The current suite of model animation test cases for these polygon algorithms consists of 500 polygons and 200 points per polygon. The polygons are generated from 18 core polygons by randomly perturbing, rotating, and permuting them. The test points are designed to stress the resolution and ray casting methods in particular. The generated test points include those that are:

- randomly (uniform) generated,
- at or near one of the vertices (multiple standard deviations used),
- exactly `BUFF` distance away from a vertex,
- on or near one of the edges (multiple standard deviations used),
- exactly `BUFF` distance away from an edge, and that
- have exactly the same or near the same x or y coordinate as a vertex.

3.3.2 Agreement Testing

The test cases generated above are generated in Python and each instance is evaluated in the software implementations in C++, Java, and Python. Inputs and outputs are then written in a text file in PVS syntax, so that PVS can be used for evaluation and comparison. A special file reader was created that evaluated each corresponding PVS function and stored any differences in another output file.

As noted above, four bugs were found (and then fixed) during the numerical evaluation stage. These are described in the following four subsections.

3.3.3 Problem 1: Loop in `closest_edge`

During numerical evaluation of the function `outside_recovery_point`, the output record of errors contained the following lines.

```
Failed Resolve OUT for (# x:=9305.5, y:=-5000.1 #).
IO reports (# x:=10001.4, y:=-5001.4 #).
Original PVS out rec is (# x:=9305.5, y:=-5002.0 #)
```

In this example, it can be seen that symbolically evaluating `outside_recovery_point` on the vector $(9305.5, -5000.1)$ returns a point near the input point. This is because the input point is very close to the last edge of the polygon. However, as can be seen from the output above that the Java implementation of this function returned $(10001.4, -5001.4)$, which is quite far (in meters) from the input point.

Recall from Section 3.1.1 that in many cases, the resolutions algorithms simply find the closest boundary point to the input position and suggest a point `ResolBUFF` away from this point, perpendicular to this edge. This requires first finding the closest edge using `closest_edge`. Upon examination, the error above was found to be caused by an incorrect implementation of the function `closest_edge`. The recursive PVS definition was proved correct, meaning that the error was in the Java implementation. The recursion over all edges from the PVS version was implemented as a for-loop in Java. Unfortunately, the highest index considered in the for loop was `p.size()-2`, which is one less than needed for correctness. This caused an incorrect result whenever closest edge to the input position was the last edge in the polygon. Thus, the line

```
for (int i = 0; i < p.size()-1; i++)
```

was changed to

```
for (int i = 0; i < p.size(); i++)
```

which fixed the problem.

3.3.4 Problem 2: Loop in `counterclockwise_corner_index`

During numerical evaluation of the function `nice_poly_2D`, which was tested on every randomly generated polygon, multiple polygons produced different answers between the C++ and PVS implementations. For these polygons, the PVS implementation of `nice_poly_2D` implied that they were well formed, while the C++ implementation implied that they were not. Examination revealed that the function `counterclockwise_corner_index` was returning an incorrect result in C++. Recall that this function returns the index of an extremal vertex of the polygon, and it is then checked that the polygon makes a left turn at this vertex. The function finds such an extremal vertex by recursively (or iteratively in C++) checking every possible index. The problem with the C++ implementation was that not every possible index was being checked, so the returned index was not necessarily that of an extremal index. This is because the highest index considered in the for loop of the function `counterclockwise_corner_index` was `p.size()-2` in the C++ implementation. Thus, the line

```
for (int i = 0; i < p.size()-1; i++)
```

was changed to

```
for (int i = 0; i < p.size(); i++)
```

which fixed the problem. The Java implementation did not have this problem. It is unclear to the developers what caused this translation error and why it is so similar to the error in the function `closest_edge` mentioned above.

3.3.5 Problem 3: Return Statement in the functions `inside_recovery_point` and `outside_recovery_point`

During numerical evaluation of `inside_recovery_point` and `outside_recovery_point`, the PVS and C++ versions produced different results for the six-point polygon with vertices at the points $(28520.0, -23520.0)$, $(28520.0, -5000.0)$, $(28520.0, 13520.0)$, $(-8520.0, -23520.0)$, $(-8520.0, -5000.0)$, and $(-8520.0, 13520.0)$, and for the input point $(-7005.4, 4020.8)$. It is important to note that this polygon does not pass the well-formed polygon test given by the function `nice_poly_2D`. Even though `inside_recovery_point` and `outside_recovery_point` should only be used in practice on polygons that pass this test, model animation of these functions for polygons that do not pass is still useful to ensure that the implemented code is faithful to the PVS specification. Indeed, using polygons that are not well-formed, as in this case, helped this problem to be found. In this example, the output text from the model animation testing produced the following lines.

```
Failed Resolve IN for (# x:=-7005.4, y:=4020.8 #).
Polygon 15. pt num 6.
PVS says (# x:=-7005.4, y:=4020.8 #)
and lang says (# x:=-8522, y:=-5000 #)
Failed Resolve OUT for (# x:=-7005.4, y:=4020.8 #).
Polygon 15. pt num 6.
PVS says (# x:=-7005.4, y:=4020.8 #)
and lang says (# x:=-8518, y:=-5000 #)
```

Thus, the inside and outside recovery functions in PVS both suggest the input point as the recovery point, indicating that no other recovery point is sufficient (because the polygon is not well-formed) and therefore the position should not be moved. The function `inside_recovery_point` (informally) works as follows (the function `outside_recovery_point` is similar).

1. Find the closest point on the boundary to the input point **s** and suggest a point *ans* `ResolBUFF` away from this boundary point, perpendicular to this edge. If this point *ans* is definitely inside the polygon, return *ans*.

2. Otherwise the point *ans* is set to another position a certain distance from $p(i)$ (closest endpoint on the closest edge to **s**) along the vector `proj_vect`$(p(i-1), p(i), p(i+1))$, where `proj_vect` is defined in Section 3.1.1. If *ans* is definitely inside, return *ans*. (See the PVS for details.)

3. If both of those fail, return the input point **s**.

The differences between the C++ and the PVS for this particular polygon were due to the fact that the input point **s** was never returned in this third step. Instead, either *ans* (from step 1) or *ans* (from step 2) was returned in every case. Upon close inspection, the final return statement, which should have returned **s** after the first steps failed, returned *ans*, with its value given by the computation in step 2. Thus, the incorrect line

```
return ans;
```

was changed to

```
return s;
```

which fixed the problem.

3.3.6 Problem 4: Geometric Condition in `proj_vect`

During numerical evaluation of the function `inside_recovery_point`, the output record of errors from the Java development contained the following lines.

```
Failed Resolve IN for (# x:=11373.5, y:=246.8 #).
Polygon 36. pt num 60.
PVS says (# x:=11640.6, y:=557.9 #)
and lang says (# x:=11373.5, y:=246.8 #) "
```

In this example, symbolically evaluating the resolution function `inside_recovery_point` in PVS on the input vector $(11373.5, 246.8)$ gave $(11640.6, 557.9)$ while the Java simply returned the input point.

Recall from Section 3.1.1 that the function `proj_vect` computes a direction vector to the *right* of the intersection of the line segments $u-v$ and $v-w$, as illustrated in Figure 3. In some cases when the input position it is near a vertex $p(i)$, `inside_recovery_point` returns a point a certain distance from $p(i)$ along the vector

$$\texttt{proj\_vect}(p(i-1), p(i), p(i+1)).$$

Upon inspection, the function `proj_vect` was implemented incorrectly in Java and C++. The second if condition "if $(w-v)\cdot(v-u)^{\perp} \leq 0$ then $(v-u)^{\perp} + (v-w)^{\perp}$" was implemented in Java and C++ as "if $(v-u)\cdot(v-u)^{\perp} \leq 0$ then $(v-u)^{\perp} + (v-w)^{\perp}$".

The condition $(w-v)\cdot(v-u)^{\perp} \leq 0$ tests whether the vector $w-v$ is to the right of $v-u$. When it is incorrectly implemented as the inequality $(v-u)\cdot(v-u)^{\perp} \leq 0$, it will always return true, because $(v-u)\cdot(v-u)^{\perp} \equiv 0$. Thus, in these developments, the line

```
if (v.Sub(u).det(v.Sub(u)) <= 0)
```

was changed to

```
    if (w.Sub(v).det(v.Sub(u)) <= 0)
```

which fixed the problem.

3.3.7 The Final Product

After fixing the four bugs mentioned in the previous subsections, the C++, Java, and Python implementations of the geo-containment algorithms passed the model animation test in the NE (Numerically Evaluated) step of MINERVA. As noted above in Section 3.3.1, the generated test cases output as text which was used by PVS for evaluation and comparison. The following is the final ouput file from these PVS tests:

```
Real time: 18h:43m:55.498s. Run time: 17659.770 sec
Lines: 806510. Records: 500. Fails: 0
```

4 Related Work

The MINERVA process is similar to model-based development techniques in that mathematical models are first developed and simulated, before an implementation is produced. Model-based commercial tools like MathWork's Simulink[5] are widely-used in the analysis and development of embedded systems. In these tools, the behavior of a system is typically specified in high-level graphical languages such as state charts. These graphical models can be translated into formalisms such as hybrid automata [3], which can be formally analyzed using various techniques (see for example [2]). Furthermore, tools like Simulink automatically generate code from these models. The generated code, which is not intended to be human readable, is usually not formally verified against high-level functional requirements.

To address the issue of unverified code generation in model-based techniques, Ryabtsev and Strichman propose a technique to verify the semantic equivalence between Simulink models and the generated C code [18]. Wang et al propose an automated credible auto-coding framework for control systems [20]. Similarly to MINERVA, this framework uses PVS to verify high level functional properties and the software verification tool Frama-C [8] to prove the correctness the generated code. The MINERVA process is less ambitions than these approaches. First, it does not aim to full correctness of the produced code, which may be challenging in the case of numerically intensive software. Second, it focuses on the development of functional components of safety-critical embedded systems. These components can be integrated as trusted modules of model-based developments. Thus, MINERVA is complementary to model-based approaches, where it offers a light weight alternative to software verification.

In the context of formal methods, tools like PVSio-web [15], which is built on top of PVS, and PetShop [17], which animates Petri nets, provide powerful features for prototyping and validating formal specifications. In [1], VDM models are animated 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 like the approach proposed in this paper.

The Numerical Evaluation (NE) step of MINERVA is similar to the technique supported by tools like QuickCheck [5] for Haskell and AutoTest [11] for Eiffel. These tools check software annotations on a set of randomly generated test cases. Similar tools exist for theorem provers [14]

[5]http://www.mathworks.com/products/simulink.

and other formal methods [21]. For example, Isabelle/HOL's Quickcheck finds counterexample to specified conjectures by random testing. However, these tools do not address the semantic gap between code and formal specifications due, for example, to numerical computations.

Concolic test [19] and other test generation techniques [4] 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.

5 Conclusion

This paper describes a software development lifecycle process called MINERVA, which stands for Mirrored Implementation Numerically Evaluated against Rigorously Verified Algorithms. The process relies on formal methods and the final product is code in a programming language that the developer chooses. Using this process, algorithms are specified and requirements are verified using an interactive theorem prover. Model animation is also used on these algorithms to extract values from them on a set of input test cases, and the output values are then compared with output values of the implementations of the algorithms in software. Thus, MINERVA is a practical way to ensure that algorithm specifications are mathematically correct and that their software implementations are faithful representations of the specifications. The MINERVA process is illustrated in this paper through its application to polygon algorithms being developed for the PolyCARP software package. PolyCARP is used inside NASA Langley's ICAROUS software package to provide geo-containment and obstacle-avoidance capabilities to small unmanned aircraft for research purposes. To illustrate the practical benefits of the process, four software bugs are shown that were found and fixed using MINERVA on the algorithms in PolyCARP.

References

[1] 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.

[2] Rajeev Alur. Formal verification of hybrid systems. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 273–278, New York, NY, USA, 2011. ACM.

[3] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of simulink/stateflow models. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 89–98, New York, NY, USA, 2008. ACM.

[4] 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.

[5] 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.

[6] María Consiglio, César Muñoz, George Hagen, Anthony Narkawicz, and Swee Balachandran. ICAROUS: Integrated Configurable Algorithms for Reliable Operations of Unmanned Systems. In *Proceedings of the 35th Digital Avionics Systems Conference (DASC 2016)*, Sacramento, California, US, September 2016.

[7] Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001.

[8] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[9] Aaron Dutle, César Muñoz, Anthony Narkawicz, and Ricky Butler. Software validation via model animation. In Jasmin Blanchette and Nikolai Kosmatov, editors, *Proceedings of the 9th International Conference on Tests & Proofs (TAP 2015)*, volume 9154 of *Lecture Notes in Computer Science*, pages 92–108, L'Aquila, Italy, July 2015. Springer.

[10] Leonard Lensink, César Muñoz, and Alwyn Goodloe. From verified models to verifiable code. Technical Memorandum NASA/TM-2009-215943, NASA, Langley Research Center, Hampton VA 23681-2199, USA, June 2009.

[11] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Yi Wei, and E. Stapf. Programs that test themselves. *Computer*, 42(9):46–55, Sept 2009.

[12] César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, and María Consiglio. DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In *Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015)*, Prague, Czech Republic, September 2015.

[13] Anthony Narkawicz and George Hagen. Algorithms for collision detection between a point and a moving polygon, with applications to aircraft weather avoidance. In *Proceedings of the AIAA Aviation Conference 2016*, Washington, DC, June 2016.

[14] Stefan Berghofer 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.

[15] Patrick Oladimeji, Paolo Masci, Paul Curzon, and Harold Thimbleby. PVSio-web: a tool for rapid prototyping device user interfaces in PVS. *Electronic Communications of the EASST*, 69, 2014.

[16] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[17] Philippe Palanque, Jean-Franois Ladry, David Navarre, and Eric Barboni. High-fidelity prototyping of interactive systems can be formal too. In JulieA. Jacko, editor, *Human-Computer Interaction. New Trends*, volume 5610 of *Lecture Notes in Computer Science*, pages 667–676. Springer Berlin Heidelberg, 2009.

[18] Michael Ryabtsev and Ofer Strichman. *Translation Validation: From Simulink to C*, pages 696–701. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[19] 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.

[20] Timothy Wang, Romain Jobredeaux, Heber Herencia, Pierre-Loïc Garoche, Arnaud Dieumegard, Éric Féron, and Marc Pantel. From design to implementation: an automated, credible autocoding chain for control systems. In *Advances in Control System Technology for Aerospace Applications*, pages 137–180. Springer Berlin Heidelberg, 2016.

[21] Wada Yusuke and Kusakabe Shigeru. Performance evaluation of a testing framework using QuickCheck and Hadoop. *IPSJ Journal*, 53(2):7p, feb 2012.