FORMAL VERIFICATION OF AN AUTONOMOUS GRASPING ALGORITHM

Virtual Conference 19–23 October 2020

Marie Farrell[1], Nikos Mavrakis[2], Clare Dixon[3], Yang Gao[4]

[1]Department of Computer Science, University of Manchester, UK, E-mail: marie.farrell@manchester.ac.uk
[2]STAR-Lab, Surrey Space Centre, University of Surrey, UK, E-mail: n.mavrakis@surrey.ac.uk
[3]Department of Computer Science, University of Manchester, UK, E-mail: clare.dixon@manchester.ac.uk
[4]STAR-Lab, Surrey Space Centre, University of Surrey, UK, E-mail: yang.gao@surrey.ac.uk

ABSTRACT

Verifying that autonomous space robotic software behaves correctly is crucial, particularly since such software tends to be mission-critical, where a software failure often equates to mission failure. Formal verification is a technique used to reason about the correctness of a software system with the output providing a proof of correctness that the software behaves as it should. In this paper, we report on our use of the Dafny program verifier to formally verify a previously developed algorithm for grasping a spacecraft motor nozzle.

1 INTRODUCTION

Removing orbital debris is an important activity to maintain easy access to space and uninterrupted orbital operations. Approximately 18% of catalogued debris consists of launch products such as spent rocket stages [1]. Active Debris Removal is the field of studying methods for removing such debris from orbit, and a variety of methods have been proposed for capturing and removal [2]. Current approaches to removing these items include the use of autonomous robotics which are equipped with an arm to capture this kind of debris [3]. A central part of the process of removing space debris is in identifying a suitable grasping point on the target surface and ensuring a stable grasp.

Autonomous robotic systems, as used in this scenario, rely on software to control their hardware components. In this setting a failure in the software could equate to complete mission failure and could create more space debris rather than reduce it as was the intention. Thus, it is crucial to ensure that the software controlling the robotic system behaves correctly.

Formal methods are a set of mathematically grounded techniques for verifying that a software system functions correctly. There are two broad categories of formal method: model-checkers which exhaustively examine the state space and theorem provers which provide a proof of correctness for the system. Formal methods are of particular use when a high degree of reliability is sought for a system that cannot be easily accessed/fixed by humans or to provide verification evidence when a system must be certified prior to use [4].

In this paper, we report on our experience of using the Dafny formal method [5] to verify an algorithm for autonomous grasping of spent apogee kick motors [3]. Originally developed by Microsoft Research, Dafny is a formal verification system which uses a theorem prover to verify properties about software [5]. Dafny has been used in several space-related applications. These include a verified component of a simulation of the Mars Curiosity rover [6] and in a case study of the Cooperative Awareness Messaging protocol for autonomous vehicles [7].

The remainder of this paper is structured as follows. In Section 2, we outline the requirements for the grasping algorithm, and we describe the basic steps in the algorithm itself along with its inputs and outputs. Section 3 describes our Dafny encoding and verification of the grasping algorithm. Section 4 discusses our approach. Finally, Section 5 concludes and outlines future research directions.

2 USE CASE: AUTONOMOUS GRASPING

The algorithm to be verified is a grasp synthesiser for the nozzle of a satellite engine. It has been tested both in simulation and in experiments with the STAR-Lab orbital testbed [8]. The details of the grasping algorithm, along with the experimental results are analytically described in [3]. A brief description of the relevant requirements and the algorithm itself is provided in this section. Terminology: this application usually involves a “chasing spacecraft” or “chaser” whose goal is to capture and remove the “target” spacecraft/debris.

2.1 Requirements for Orbital Debris Capturing

Space debris capturing is an inherently novel and unknown type of operation, with limited standardisation worldwide. Nevertheless, many

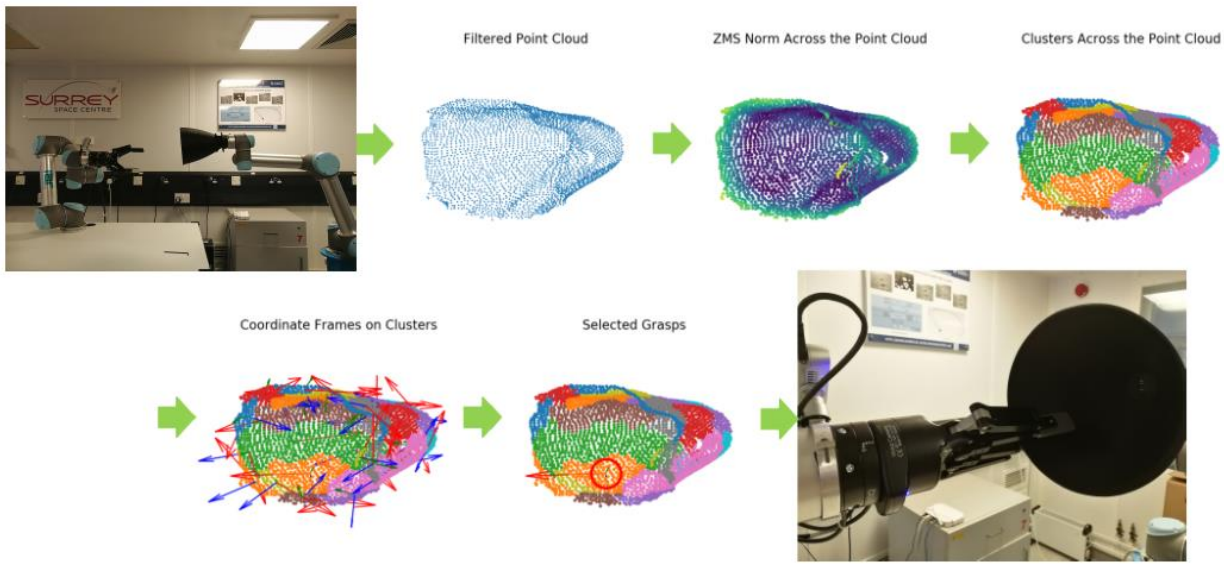

Figure 1 Execution of the grasping algorithm from point cloud to the selected grasp for a real robot setup.

engineering challenges have been defined and must be taken into consideration in every proposed solution:

- The target state and properties must be known in advance, or otherwise identified. Examples of identifiable properties include the target's pose, angular rate, and dynamic parameters.
- In the case where a pre-determined capture point exists, the chasing spacecraft must identify it. If such capture point does not exist, the spacecraft must *synthesize* and identify a grasping point based on the calculated target properties.
- The contact on the target must be executed in a compliant way, so that the target is not pushed away, and the spacecraft's base remains relatively undisturbed.
- The chasing spacecraft must implement a series of controllers for all capturing phases (rendezvous, proximity, contact, post-contact), and have the capability of switching between them.

In this paper, we draw inspiration from the second challenge and define the basic set of requirements to be verified as shown in Tab. 1. As our method in [3] already outputs a set of grasping points on a nozzle surface, we wish to verify its three most important properties, i.e. that the final grasp exists out of the group of returned ones, that it satisfies the reachability criteria defined in [3], and that if no suitable grasp is found then no grasp is chosen. Various sub-requirements related to the Dafny model are also verified and will be outlined in Section 3.

Table 1 Basic requirements for grasping algorithm.

ID	Description
R1	The chosen grasp shall be optimal based on predefined criteria.
R2	The optimal grasp shall be chosen from the list of computed grasps.
R3	If there are no valid grasps computed, then none shall be chosen.

2.2 The Grasping Algorithm

Fig. 1 illustrates the intermediate steps of the grasping algorithm which is implemented in Python. At first, a point cloud of the target nozzle is acquired. To make further processing faster, the point cloud is downsampled through voxelisation. A geometric filter is applied to filter out noise and points without many neighbours. The surface is then characterized, according to its smoothness, by calculating the Zero Moment Shift (ZMS) vector for every point. The ZMS is a metric for the distance of each point to the average of its local neighbourhood [9].

Then, we form a 4D feature vector for every point, composed of its 3D coordinates and the vector norm of its ZMS. The vectors are used as input to a clustering algorithm that separates the nozzle surface into graspable patches. To turn the patches into

grasping poses (called "grasps"), Principal Component Analysis (PCA) is applied to each patch to extract a 3D coordinate frame for the gripper pad to be placed on. As some grasps may not be reachable, the grasps are evaluated according to pre-defined reachability criteria, and the best grasping pose is selected [3]. This corresponds to the grasp whose area score is above a certain threshold and has the smallest angle score.

In order to verify the algorithm, we first translate these intermediate steps into pseudocode description. The pseudocode decomposes the functionality of the algorithm into modules with pre-defined inputs and outputs. This helps us to design pre- and post-conditions both for each module and the algorithm, facilitating the verification of numerous properties. The pseudocode is given in Algorithm 1.

The grasp generation process is explained as follows:

Line 2: The *removeDepth* function gets as input a pointcloud *p* and a depth threshold *t*. It removes any points whose distance from the camera is greater than the threshold, *t*. It returns a thresholded point cloud p_r.

Line 3: The *downSample* function gets as input the thresholded point cloud p_r and a voxel size *v*. It splits the point cloud into 3D geometric voxels and averages the points in each voxel. The result is a downsampled point cloud p_d.

Line 4: The *filter* function gets as input the downsampled point cloud p_d, a filtering radius r_f, and an integer n_p. It removes the points that have less than n_p neighbouring points within a radius of r_f. The result is a filtered point cloud p_f.

Line 5: The *calculateZMS* function gets as input the filtered point cloud p_f and a radius r_b. It calculates the 3D vector and L_2 norm for each point in the cloud, around a neighbourhood defined by r_b (i.e. the Euclidean distance of the point to the mean of its neighbourhood). The result is a vector of norms L_z.

Line 6: The *formClusterInput* function gets as input the filtered point cloud p_f and the vector of norms L_z. It returns a concatenated feature array x_p composed of the 3D coordinates of each point and the corresponding ZMS norm.

Line 7: The *clustering* function gets as input the feature array x_p and the number of clusters n_c. It clusters the point cloud in graspable patches, according to their local geometry and surface smoothness. The result is a set of *clusters*.

Data:
p: original point cloud
p_r: thresholded point cloud
p_f: filtered point cloud
L_z: L2 norm vector of ZMS on each point
x_p: Input matrix for clustering
t: point depth threshold
v: voxel size
p_d: downsampled point cloud
r_f: filtering radius
n_p: max number of points within the filtering radius
r_b: radius for ZMS calculation
n_c: number of clusters
clusters: list of clusters
grasps list of grasping points
dmax: max reachability limit
dmin: min reachability limit
ga: gripper pad area
θ: limit angle
Result: A grasping pose *gp*, corresponding to the optimal grasp

```
1  begin
2      p_r ← removeDepth(p, t) ;
3      p_d ← downSample(p_r, v) ;
4      p_f ← filter(p_d, n_p, r_f) ;
5      L_z ← calculateZMS(p_f, r_b) ;
6      x_p ← formClusterInput(L_z, p_f);
7      clusters ← clustering(x_p, n_c) ;
8      grasps ← PCA(clusters);
9      gp ←
         getOptimalGrasp(grasps, dmax, dmin, ga, θ);
10     return gp
11 end
```

Algorithm 1 An outline of the steps used in the grasping algorithm.

Line 8: The *PCA* function gets as input the set of *clusters*. It applies Principal Component Analysis on the clusters to extract a 3D coordinate frame for each patch. The result is a set of grasping poses, *grasps*.

Line 9: The *getOptimalGrasp* function gets as input the set of *grasps*, an upper and lower depth limit, *dmax* and *dmin*, a number, *ga*, representing a threshold area value, and a limit angle θ. It selects as optimal the grasp that has: distance from the robot between *dmax* and *dmin*, cluster area greater than *ga*, and yaw angle between $-\theta$ and θ. The result is the final optimal grasp *gp*. If more than one grasp satisfies the reachability criteria, one is selected at random. If no grasps are generated or none satisfy the criteria then no grasp is returned.

These are the basic steps in the grasping algorithm. In the next section, we describe how we construct a corresponding Dafny encoding of this algorithm and verify the requirements.

3 VERIFICATION WITH DAFNY

Dafny is a formal verification system that is used in the static verification of functional program correctness. Users provide specification constructs e.g. pre-/post-conditions, loop invariants and variants [5]. Programs are translated into the Boogie intermediate verification language [10] and then the Z3 automated theorem prover discharges the associated proof obligations [11] that were generated[1]. The basic paradigm in formal verification is to demonstrate that, if the program is executed in a state which satisfies its pre-condition(s) then it will terminate in a state which satisfies its post-condition(s).

The basic structure of a method in Dafny is outlined in Fig. 2. Here, the **requires** keyword on line 2 is used to indicate the pre-condition for the method, the **modifies** keyword on line 3 specifies which of the input variables the method is allowed to modify and the **ensures** keyword on line 4 accounts for the method's post-condition. The user specifies the loop **invariant** on line 7 which is used by the underlying SMT solver to prove that the post-condition (line 4) is preserved and the **decreases** clause on line 8 corresponds to a loop variant for proving loop termination.

```
1  method myMethod(x: int) returns (y: int)
2  requires ...
3  modifies ...
4  ensures ...
5  {
6    while (i<x) //this is a comment
7    invariant ...
8    decreases ...
9    {
10     ...
11   }
12 }
```

Figure 2 The basic structure of a method with specification constructs in Dafny.

3.1 Our Approach

For the grasping algorithm as outlined in Section 2, we devised a formal model of the grasp planning algorithm in Dafny including the definition of two particular methods. One which captures the functionality of Algorithm 1 and another which specifically focuses on selecting the optimal grasp.

We also had to model a series of helper functions since the Python implementation of the grasping algorithm uses library functions which are not available in Dafny. For the purposes of verification, we specify pre-/post-conditions for each of these methods.

We specifically focus on verifying that the chosen grasp is valid, correct, and optimal with respect to the defined criteria [3]. We also verify that the helper functions behave as expected, as well as the usual suite of standard program correctness properties (e.g. loop termination, etc.).

We encoded our basic types using tuples in Dafny as shown in Fig. 3. In particular, a **Point** is a tuple with three real number elements (lines 1-2). A **Grasp** (lines 3-5) has seven real number elements. The first 3 elements (x, y, z) encode the distance of the grasp coordinate frame from the camera coordinate frame (which are known after calibration). The last 4 elements (qx, qy, qz, qw) are a quaternion, and they encode the orientation of the grasp frame with respect to the camera frame. Finally, a **Score** has three real number elements. The first is an area score (this will be *1.0* if the calculated area is above the area threshold, *0.0* otherwise), the second is an angle score which is the score for the cluster and finally the index stores the index in the sequence of calculated grasps which corresponds to this score.

```
1  datatype Point =
2    Point(x:real, y:real, z:real)
3  datatype Grasp =
4    Grasp(x:real, y:real, z:real, qx:real,
5          qy:real, qz:real, qw:real)
6  datatype Score =
7    Score(areaScore:real, angleScore:real,
8          index:int)
```

Figure 3 The basic datatypes for storing points, grasps and scores in our Dafny model.

3.2 Modelling Algorithm 1

The method presented in Fig. 4 accounts for the main functionality of the grasp planning algorithm. This corresponds to Algorithm 1 as outlined earlier. In particular:

> **Lines 1-3:** This method takes a point cloud, p, as input, it outputs the generated sequence of grasps, the optimalgrasp and a boolean flag

[1] A proof obligation is a theorem describing a property that must be proven for the software system to be formally verified.

```
1  method graspplanner(p:array<Point>)
2   returns(grasps:seq<Grasp>, optimalgrasp:Grasp,
3           nograsp:bool)
4  requires p.Length > 0;
5  modifies p;
6  ensures |grasps|>0  ⇒  ∃ i · 0≤i<|grasps|
7          ∧ grasps[i] = optimalgrasp
8          ∧ nograsp = false;
9  ensures |grasps| = 0  ⇒  nograsp = true;
10 {
11   var t := 1.0; //threshold
12   var v := 0.03 as real; //voxel size
13   var np := 16; //number of neighbouring points
14   var rf := 0.05 as real; //filtering radius
15   var rb := 0.5 as real; //ball radius
16   var a := 3.0 as real; //scaling parameter
17   var nc := 30; //number of clusters.
18   var dmin := 0.05 as real; //min cluster depth
19   var dmax := 0.15 as real; //max cluster depth
20   var ga := 0.0011266 as real;
21     //gripper pad area
22   var thetamin, thetamax := -30, 30;
23     //edge angle limits
24   var pcopy := p; //copy of p
25   nograsp := false;
26
27   pcopy := removeDepth(pcopy,t);
28   //remove distant points from pcopy
29   pcopy := downSample(pcopy,v);
30   //voxel representation of pcopy (low res).
31   pcopy := filter(pcopy,nb,rf); /
32   //remove noise and speckles
33
34   var z := Point(0.0,0.0,0.0);
35   //default value, updated later.
36   var xp:= [];
37   var j := 0;
38   while (j<pcopy.Length)
39   decreases pcopy.Length - j;
40   {
41     z := calculateZMS(rb, pcopy, pcopy[j] );
42     var np := vectorNorm(z);
43     xp := xp +
44           [pcopy[j].x, pcopy[j].y, pcopy[j].z,
45             a*(np.Floor as real)];
46     j:= j+1;
47   }
48   var clusters := clustering(xp,nc);
49   grasps := [];
50   var scores : seq<Score>;
51   scores:=[];
52   var i := 0;
53   while(i<clusters.Length)
54   invariant 0≤ i ≤ clusters.Length;
55   invariant |scores| ≤ |grasps|;
56   invariant ∀ s · 0≤s<|scores|
57               ⇒ 0<scores[s].index<|grasps|;
58   invariant ∀ s · 0≤s<|scores|
59               ⇒ scores[s].areaScore=1.0;
60   decreases clusters.Length - i;
61   {
62     var mei := PCA(clusters[i]);
63     var centroid := mei[0];
64
65     if(dmin≤centroid.z as real
66         ∧ centroid.z as real ≤dmax)
67     {
68       var area := calculateArea(clusters[i]);
69       var theta :=
70        calculateXangle([mei[1],mei[2],mei[3]]);
71
72       if(area as real ≥ ga){
73         scores := scores +
74                   [Score(1.0,theta,|grasps|)];
75       }
76       grasps := grasps +
77         [Grasp(centroid.x, centroid.y,
78           centroid.z, mei[1].x, mei[2].y,
79           mei[3].z, theta)];
80     }
81     i := i+1;
82   }
83   if (|grasps|>0 ∧|scores| >0){
84     var optimalscoreindex;
85     optimalscoreindex, optimalgrasp :=
86       selectOptimalGrasp(grasps, scores);
87   }
88   else if(|grasps|>0 ∧ |scores|=0){
89     optimalgrasp :=  grasps[0];
90   }
91   else if (|grasps|=0){
92     nograsp := true;
93   }
94   return grasps, optimalgrasp, nograsp;
95 }
```

Figure 4 Dafny encoding of the graspplanner method.

which is used to alert the user if no suitable grasp was calculated.

Lines 4-9: The pre-condition on line 4 specifies that the input array must not be empty and the **modifies** clause on line 5 allows us to modify the input. The post-condition on line 8 **ensures** that the optimal grasp is indeed a grasp that was calculated. Then the post-condition on line 9 specifies that if there are no valid grasps returned then the nograsp boolean is set to true.

Lines 10-37: These variables correspond to those that were outlined in Algorithm 1. We have tried to maintain the nomenclature where possible but for implementation purposes we had to include additional variables. For example, we include pcopy to keep track of the updates that we make to the input point cloud, p. In particular, we update pcopy when we call the methods on lines 27-32 which correspond to the steps on lines 2-4 of Algorithm 1. We also decomposed θ as used in Algorithm 1 into thetamin and thetamax for implementation purposes on line 22 of Fig. 4.

Lines 38-48: This while loop captures the functionality of lines 5-7 of Algorithm 1 by calculating the ZMS, forming the correct input to the clustering method and then executing clustering with the appropriate inputs. Note that the **decreases** clause on line 39 is used to prove loop termination[2] which is a basic program correctness property that Dafny requires us to preserve.

Lines 49-62: This part of the method calculates the potential grasps and assigns a score for each one using the criteria devised in [3]. We capture line 8 of Algorithm 1, which invokes PCA, here on line 62. The loop invariants shown here on lines 54-59 ensure that the loop variable remains within appropriate bounds (line 54), the correct number of scores are placed into the sequence of scores (line 55), that each score is matched to the corresponding grasp (lines 56-57), and that only scores with an appropriate area component are stored (lines 58-59). As above, line 60 is used to prove loop termination.

Lines 63-95: This part of the method corresponds to the function called on line 9 of Algorithm 1. In particular, lines 63-82 implement the score calculation. Once the scores and grasps have been

[2] Each iteration of the loop decreases the value of pcopy.Length - j until it reaches zero.

calculated, we then call the selectOptimalGrasp method which chooses the optimal grasp based on the criteria in [3] (lines 83-87). In case no grasp meets the ideal score, we simply return the first one in the sequence (lines 88-90) and if no grasps could be calculated then we toggle the nograsp boolean flag (lines 91-93).

3.3 Selecting the Optimal Grasp

Fig. 5 contains the Dafny model of the method for selecting the optimal grasp based on the calculated scores. We describe its functionality as follows:

Lines 1-4: Since it is important to ensure that the chosen grasp is optimal, this method takes the sequence of computed grasps and the associated scores as input and outputs the index of the optimal score (optimalscoreindex) along with the corresponding optimalgrasp.

Lines 5-8: These are the pre-conditions for this method. The first pre-condition (lines 5-6) states that every index in the sequence of scores must be a valid index in the sequence of grasps. This is necessary so that we do not allow for the scores sequence to refer to a grasp that is not present in the sequence of grasps. The pre-conditions on lines 7 and 8 require that neither of the inputs are empty sequences and that there are at least as many scores as there are grasps in the input. Since this method is called by the graspplanner method (Fig. 4), we must verify that these pre-conditions are met. This is achieved via the loop invariants on lines 55-57 of Fig. 4.

Lines 9-14: These post-conditions ensure that the selected optimal grasp is indeed present in the input sequence of grasps (lines 9-10) and that the optimalgrasp is indeed optimal because it is the one with the smallest angle score (lines 12-14). We also ensure that the corresponding index is a valid index in the scores sequence (line 11).

Lines 15-18: Declare and initialise the necessary variables. We use minscore to keep track of the best score found so far.

Lines 19-40: The while loop on lines 19-38 is used to step through the scores sequence and keep track of the best score so far. Then we return the optimalgrasp as the one in the sequence of grasps at the identified index (line 39). The invariants on lines 20-29 ensure that the loop behaves correctly and are used by the prover to verify the earlier post-conditions (on lines 9-14). The loop variant on line 30 is used to prove loop termination.

```
1  method selectOptimalGrasp(grasps: seq<Grasp>,
2                            scores: seq<Score>)
3   returns (optimalscoreindex: int,
4            optimalgrasp: Grasp)
5  requires ∀ s · 0≤s<|scores|
6           ⇒ 0≤scores[s].index<|grasps|;
7  requires |scores|>0 ∧ |grasps|>0;
8  requires |grasps|≥|scores|;
9  ensures ∃ i · 0≤i<|grasps|
10          ∧ grasps[i]=optimalgrasp;
11 ensures 0≤optimalscoreindex<|scores|;
12 ensures ∀ s · 0≤s<|scores|
13          ⇒ scores[optimalscoreindex].angleScore
14             ≤ scores[s].angleScore;
15 {
16   var minscore := scores[0];
17   var i := 0;
18   optimalscoreindex := 0;
19   while(i<|scores|)
20   invariant 0≤i≤|scores|;
21   invariant 0≤minscore.index<|grasps|;
22   invariant ∀ s · 0≤s<i
23              ⇒ minscore.angleScore
24                ≤ scores[s].angleScore;
25   invariant 0≤optimalscoreindex<|scores|;
26   invariant scores[optimalscoreindex]=minscore;
27   invariant ∀ s · 0≤s<i ⇒
28             scores[optimalscoreindex].angleScore
29               ≤ scores[s].angleScore;
30   decreases |scores| - i;
31   {
32     if(scores[i].angleScore≤minscore.angleScore)
33     {
34       minscore := scores[i];
35       optimalscoreindex := i;
36     }
37     i:= i+1;
38   }
39   optimalgrasp := grasps[minscore.index];
40 }
```

Figure 5 Dafny encoding of the selectOptimalGrasp method.

3.4 Helper/Library Functions

In order to adequately model the algorithm, we also constructed simplified Dafny models for most of the helper/library functions that are used in the Python implementation of the algorithm. These include removeDepth, downSample, filter, calculateZMS, vectorNorm, clustering, PCA, calculateArea and calculateXangle.

As an example, we provide the corresponding Dafny method for removeDepth in Fig. 6. This method takes a point cloud, p, and a real number threshold, t, as input. It then outputs a point cloud, pr, which is the filtered version of the input point cloud with all points whose z-value is above the threshold removed. We verified the correctness of this method and specified that it behaves correctly using the pre-/post-conditions on lines 3-6 and via the associated loop invariants (lines 12-17 and lines 30-33).

3.5 Verifying the Requirements

We listed three requirements that we wished to show in Tab. 1. We ensure that these are preserved as follows:

R1: *The chosen grasp shall be optimal based on predefined criteria.*

```
1  method removeDepth(p: array<Point>, t: real)
2    returns (pr: array<Point>)
3  requires 0 < p.Length;
4  ensures pr.Length=0
5            ⇒ ∀ m · 0≤m<p.Length ⇒ p[m].z>t;
6  ensures pr.Length ≤ p.Length;
7  {
8    var k := 0;
9    var prsize := 0;
10
11   while(k<p.Length)
12   invariant 0≤k≤p.Length;
13   invariant prsize = 0
14               ⇒ ∀ m · 0≤m<k  ⇒  p[m].z>t;
15   invariant prsize > 0
16               ⇒ ∃ m · 0≤m<k ∧ p[m].z≤t;
17   invariant prsize ≤k≤p.Length;
18   decreases p.Length - k;
19   {
20     if(p[k].z ≤ t){
21       prsize := prsize + 1;
22     }
23     k := k + 1;
24   }
25   pr := new Point[prsize];
26   k:=0;
27   var c := 0;
28
29   while(k<p.Length ∧  c<pr.Length)
30   invariant 0≤c≤pr.Length;
31   invariant 0≤k≤p.Length;
32   invariant ∀ i · 0≤i<c  ⇒  pr[i].z≤t;
33   invariant ∀ m · 0≤m<c  ⇒  pr[m].z≤t;
34   decreases p.Length -k;
35   {
36     if(p[k].z ≤ t){
37       pr[c] := p[k];
38       c := c + 1;
39       }
40     k := k+1;
41   }
42   return pr;
43 }
```

Figure 6 Dafny encoding of the removeDepth method.

The predefined criteria instruct us that the optimal grasp is one with an area score greater than a particular threshold and that it is the grasp with the smallest angle score. We verified the area part of this requirement using the loop invariant on lines 58-59 of Fig. 4. This loop invariant specifies that only grasps whose area score is equal to *1.0*, meaning that it is above the pre-defined threshold (if-statement on lines 72-75 of Fig. 4), have their corresponding score considered. We ensure that the angle part of the criteria is met by verifying the post-condition on lines 12-14 of Fig. 5 which is supported by the invariants on lines 20-29 of Fig. 5.

R2: *The optimal grasp shall be chosen from the list of computed grasps.*

This requirement is specified as a post-condition on lines 6-8 of Fig. 4 and lines 9-10 of Fig. 5. In fact, once the latter was verified it would have been used by the underlying theorem prover to verify that the former was preserved.

R3: *If there are no valid grasps computed then none shall be chosen.*

This requirement was specified as a post-condition on line 9 of Fig.4.

We also verified a number of properties that were not explicitly listed as requirements in Tab. 1. These include properties about how the scores were calculated which are expressed on lines 56-59 of Fig. 4 and, on lines 12-14 and 22-29 of Fig. 5. We also verified basic requirements about the library/helper functions, an example of these can be seen on lines 4-5, 13-16 and 32-33 of Fig. 6.

Overall we were able to discharge all proof obligations automatically using version 2.3 of Dafny in version 1.48 of Visual Studio Code on Ubuntu 18.04.

4 DISCUSSION

A large variety of methods are typically used for verification in robotics [4]. In our case, we selected the use of the Dafny language, as its similarity to general coding paradigms and its syntax enable easier translation of the actual code used in the grasping algorithm to a Dafny program [12].

Ideally a formal model would be developed alongside the implemented algorithm. However, in this case the algorithm was devised beforehand which usually complicates the verification task [4]. Fortunately, the original grasping algorithm was created with a high degree of modularity, and this enabled us to easily translate each module to Dafny, and to specify and verify the properties for each module individually, as well as for the entire algorithm.

Crucially, Dafny's specification constructs and executable code are both written using the same syntax making it easy to communicate to non-expert users. This was useful since the requirements, as described in Tab. 1, were english-language descriptions which can often be difficult to formalise in a way that is amenable to formal verification. However, we were able to construct Dafny-friendly specifications for these requirements alongside the developers of the original algorithm since the syntax was easy to understand.

From the perspective of the developers, the process of verification with Dafny is beneficial because it uses pre-conditions for the final source code variables (i.e. assumptions on the variables that need to hold for the post-conditions to hold). Those pre-conditions, specified in Dafny, can be directly implemented in the source code to make sure that it operates in accordance with the Dafny model. This speeds up the development progress and integrates the verification process in the source code, making the final program more trustworthy.

Other desirable requirements to verify were related to temporal aspects. For example: *The optimal grasp*

shall be returned within 3 seconds. However, Dafny does not support the verification of temporal properties. As such, future work includes investigating how another formal verification tool could be used to verify these properties in a complimentary fashion to our Dafny model presented in this paper.

We note that, the Dafny tool support available in Visual Studio Code [13] and on the rise4fun website[3] was useful, although the error messages could sometimes be unclear. This is a usual barrier to usability for formal methods. Another issue that we had with Dafny was that we had to create copies of some variables in order to update their contents even though we used the **modifies** clause. However, it is not clear whether this is a peculiarity of the Dafny language itself or whether there was a bug in the Dafny distribution that we used.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we described our use of the Dafny formal method to verify the correctness of an autonomous grasping algorithm for space debris removal. Future work includes providing more detailed encodings for the helper functions that are used in the Dafny model and verifying that they function correctly. We also intend to derive runtime monitors from our Dafny model which can be used to monitor the algorithm at runtime. Finally, we wish to devise and verify capturing requirements that are in accordance with international standards and regulations for orbital robotics [14]. This will enable us to assess the readiness of this grasping algorithm for use in real orbital robotics scenarios.

References

[1] European Space Agency, "About Space Debris," [Online]. Available: https://www.esa.int/Safety_Security/Space_Debris/About_space_debris.

[2] M. Shan, J. Guo and E. Gill, "Review and comparison of active space debris capturing and removal methods," *Progress in Aerospace Sciences,* vol. 80, pp. 18-32, 2016.

[3] N. Mavrakis and Y. Gao, "Visually Guided Robot Grasping of a Spacecraft's Apogee Kick Motor," in *Symposium on Advanced Space Technologies In Robotics and Automation*, 2019.

[4] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon and M. Fisher, "Formal Specification and Verification of Autonomous Robotic Systems: A survey," *ACM Computing Surveys,* vol. 52, no. 5, pp. 1-41, 2019.

[5] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Logic for Programming Artificial Intelligence and Reasoning*, 2010.

[6] R. C. Cardoso, M. Farrell, M. Luckcuck, A. Ferrando and M. Fisher, "Heterogeneous Verification of an Autonomous Curiosity Rover," in *NASA Formal Methods Symposium*, 2020.

[7] M. Farrell, M. Bradbury, H. Yuan, M. Fisher, L. A. Dennis, C. Dixon and C. Maple, "Using Threat Analysis Techniques to Guide Formal Verification: A Case Study of Cooperative Awareness Messages," in *Software Engineering and Formal Methods*, 2019.

[8] Z. Hao, N. Mavrakis, P. Proenca, R. Gillham Darnley, S. Fallah, M. Sweeting and Y. Gao, "Ground-Based High-DOF AI And Robotics Demonstrator For In-Orbit Space Optical Telescope Assembly," in *Congress IAC 19-paper archive*, 2019.

[9] U. Clarenz, M. Rumpf and A. Telea, "Robust Feature Detection and Local Classification for Surfaces Based on Moment Analysis," *IEEE Transactions on Visualization and Computer Graphics,* vol. 10, no. 5, pp. 516-524, 2004.

[10] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*, 2005.

[11] L. De Moura and N. Bjorner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[12] K. R. M. Leino, "Accessible Software Verification with Dafny," *IEEE Software,* vol. 34, no. 6, pp. 94-97, 2017.

[13] R. Krucker and M. Schaden, "Visual Studio Code Integration for the Dafny Language and Program Verifier," HSR Hochschule fur Technik Rapperswil, 2017.

[14] ISS MCB, T, *International Deep Space Interoperability Standards-Draft C,* International Space Station Multilateral Coordination Board, 2018.

[3] https://rise4fun.com/Dafny/