CERTIFIED FUNCTIONS FOR MESH GENERATION

Andrey N. Chernikov

Old Dominion University, Norfolk, VA, U.S.A. achernik@odu.edu

ABSTRACT

Formal methods allow for building correct-by-construction software with provable guarantees. The formal development presented here resulted in certified executable functions for mesh generation. The term certified means that their correctness is established via an artifact, or certificate, which is a statement of these functions in a formal language along with the proofs of their correctness. The term is meaningful only when qualified by a specific set of properties that are proven. This manuscript elaborates on the precise statements of the properties being proven and their role in an implementation of a version of the Isosurface Stuffing algorithm by Labelle and Shewchuk. This work makes use of the Calculus of Inductive Constructions for defining executable functions, stating their properties, and proving these properties via a direct examination of these functions (the property of liveness). The certificate is made available for inspection and execution using the Coq proof assistant.

Keywords: guaranteed quality mesh generation, formal methods, correct-by-construction, certified software

1. INTRODUCTION

Formal methods have undergone a rapid development in recent years. There are a number of reasons for this phenomenon. First, as algorithms and their software implementations are becoming increasingly sophisticated, they require a corresponding increase in the efforts to argue about the properties of the results they produce. For a number of complex algorithms, like unstructured guaranteed quality mesh generation, the complexity of stating and proving their specifications necessitates multiple manuscripts and pushes the limits of the traditional paper-and-pencil approach. Second, parallel algorithms, which have been developed for virtually all areas of computing, often bear another order-of-magnitude intricacy factor over the corresponding sequential algorithms due to numerous ways a parallel system can process and move data. Third, as automated systems are becoming increasingly integrated, a failure of one component has ramifications for the whole software-hardware system and can lead to costly and/or dangerous consequences. A number of major organizations, including Amazon [1], Microsoft [2], Twitter [3], Intel [4], MIT [5, 6], and Inria [7, 8], have made progress in the development and the use of formal methods and functional programming towards managing software correctness and maintainability.

This work is the second formal verification in the area of mesh generation, to the best of the author's knowledge, that exhibits the property of *liveness* [9], i.e., being connected to the implementation via machine-checked proofs. The property of liveness increases the level of confidence in the correctness, as the proofs make use of the properties of the actual operations used in the code, not just abstracted models of those operations. The first work was published by Chernikov and Xu [10] on the correctness of a version of a Marching Cubes algorithm [11]. The present work proves certain correctness properties of the Isosurface Stuffing algorithm [12], and also differs from the previous [10] in two major aspects. The first one is that it works with tetrahedra, not just triangular faces. The second difference is in the proof approach. The previous work [10] used proofs via computation, i.e., the properties being proven were embedded in the functions computed as part of the proof. The present work sep-

arates the proofs into two parts: propositional specifications that state the properties being proven, and the actual proofs that establish that these specifications hold. As a result, the present approach is easier to read and maintain. Another known work, in formally proving properties of a plane Delaunay triangulation algorithm, is that of Dufourd and Bertot [13]. The work [13] does not exhibit the property of liveness as it relies only on axiomatization, not implementation, of real numbers. The underlying number type in the present work is integers, which is implemented in the standard library of the Coq environment [8] being used, and suitable due to the lattice-based nature of the algorithm being studied. When required by the algorithm, the intersections with the surface are computed using integer arithmetic to within a finite precision.

The development presented herein resulted in certified executable functions for mesh generation. The term *certified* means that their correctness is established via an artifact, or *certificate*, which is a statement of these functions in a formal language along with the proofs of their correctness. The term is meaningful only when qualified by a specific set of properties that are proven. The functions whose properties are being proven and the precise formal statements of their certified properties are elaborated further down in this manuscript. The certificate in the form of an accompanying Coq script is made available [14]. As can be seen from the Venn diagram in Figure 1, only a subset of the entire implementation is involved in the proof of the properties stated below. A certificate with a formal proof of another set of properties may involve a different subset of functions.

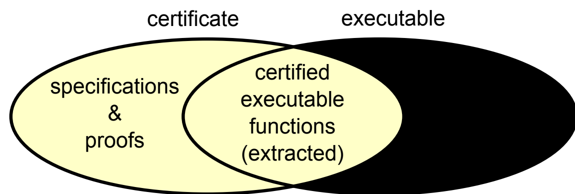

Figure 1: A Venn diagram for the components of certified executable software. The parts described here are in yellow.

Published in 2007 by Labelle and Shewchuk [12], the Isosurface Stuffing (IS) algorithm solved a longstanding problem in tetrahedral meshing of geometrically complex shapes, that of achieving practically significant guaranteed dihedral angle bounds. Among other favorable properties of this algorithm are its fast execution, mesh gradation, placement of mesh boundary vertices exactly (to within a tolerance) onto the domain boundaries, as well as certain fidelity guarantees. Conceptually, the IS algorithm is similar to the well-known Marching Cubes (MC) algorithm [11]. Both algorithms employ a regular cubical grid to cover the region of interest. The IS algorithm uses a body centered cubic (BCC) lattice that is tessellated into a structured tetrahedral grid. Both the MC and the IS algorithms evaluate each vertex of the corresponding grid using a given *cut* function, which returns one of three values, corresponding to the vertex being inside, outside, or on the surface of the shape defined by this function. Depending on the local combination of these values in the vertices of a cube, the MC algorithm outputs one of the predetermined stencils, which is a set of triangles that approximates the surface inside this cube. The IS algorithm evaluates the cut function in the vertices of a structured tetrahedron and outputs one of predefined sets of stencil tetrahedra (referred to as unstructured below). Both algorithms also make use of approximated locations of zero values of the cut function along the edges of the respective cubes or tetrahedra.

What distinguishes the IS algorithm from the MC and its variations is the presence of proofs that guarantee *a priori*, i.e., before the execution, that the dihedral and certain other angles will be bounded by known constant values. Due to a large number of cases to examine and, most importantly, to the unknown locations of the zero values within respective edges, these proofs would be next to impossible to construct manually. Instead, the IS authors implemented and described a computer-assisted approach where the computation of the angle bounds is done in software with the use of interval arithmetic and of recursive bisection of the parameter space. What was not described, however, is an *a priori* proof of correct connectivity between the adjacent tetrahedral stencils. Such a proof is important as a precedent is known with the original MC algorithm, wherein symmetry was incorrectly used to reduce the number of stencils, thus causing topological holes in the approximated surface [10]. The present work offers a formal statement of the connectivity properties of the IS stencils along with the proofs that these properties hold.

Figure 2 shows two example meshes obtained with the certified implementation discussed here. The implementation uses the same stencils in the uniformly refined area next to the domain boundary as the original IS algorithm. However, unlike the IS algorithm, it does not implement the octree optimization with additional stencils that allow for fewer tetrahedra in the interior of the domain by omitting the creation of certain octree leaves. These additional octree stencils are left for future work.

The number of IS connectivity cases to be verified, when unfolded, grows rapidly. Every two adjacent structured tetrahedra that share a triangular face in-

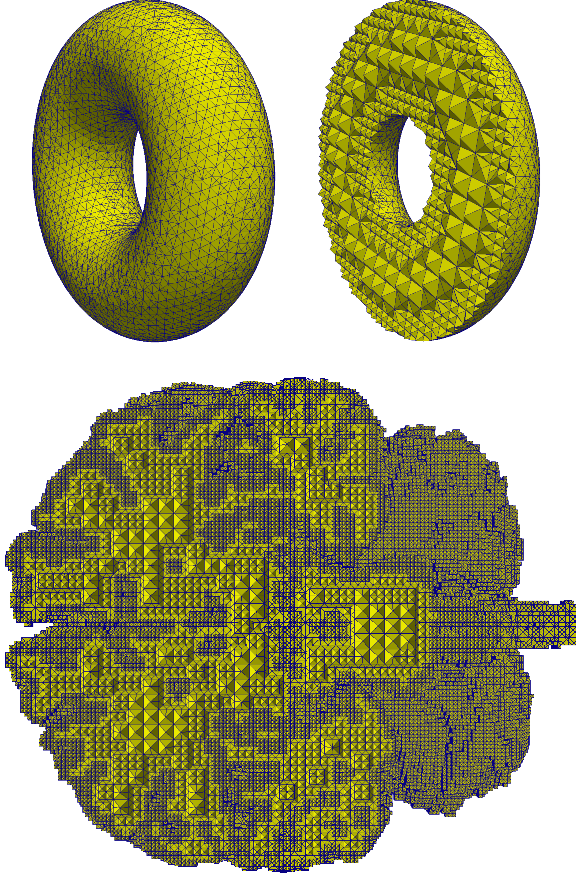

Figure 2: Top: the surface and a cross-section of a mesh of a torus defined by an analytical function (61,528 tetrahedra, 20.2° minimum output dihedral angle). Bottom: a cross-section of a mesh of a human brain atlas [15] defined by an image (9,788,808 tetrahedra, 30° minimum output dihedral angle).

volve 5 vertices (see Figure 3) and, since each vertex can evaluate the cut function to one of 3 values, the number of 5-vertex sign assignments is $3^5 = 243$. If a higher confidence in the stencils is aimed at, as is the case in the proofs that follow, each of the four faces of a structured tetrahedron needs to be examined separately, which leads to a multiplier of 4: $243 \cdot 4 = 972$. Also, if formulas for each case of a translation and a rotation of a structured tetrahedron differ, these formulas need to be looked at separately as well. In this development, 12 variations of a formula are used, which raises the number of cases to $972 \cdot 12 = 11,664$. Moreover, each case involves several (0 to 3 on each side of the shared face) unstructured tetrahedra that subdivide either of the adjacent structured tetrahedra, and every edge of each of these unstructured tetrahedra needs to be examined with respect to the subdivision of the shared face from the other side. A manual enumeration and examination of all such cases becomes highly time-consuming and error-prone. Another important consideration is the ability to modify and extend this algorithm without the need for a repeated manual analysis. Once an automatic (i.e., not depending on the stencils) verification procedure is developed, it can be trivially applied to another set of stencils.

As an example, Figure 3 shows two structured tetrahedra, ($f1$, $f2$, $f3$, p) and ($f1$, $f2$, $f3$, q), that share face ($f1$, $f2$, $f3$). The symbols in vertex positions show the signs of the cut function: "+" symbolizing the vertex being inside the domain, "−" being outside, and "0" lying on the boundary. The signs of vertices $f1$, $f2$, $f3$, p, and q are computed by direct evaluation of the cut function. The positions of vertices u, v, and w are computed by iterative bisection of the edges (q, $f1$), ($f3$, $f1$), and ($f2$, $f1$), respectively, in order to find (to within a tolerance) the zero values of the cut function, i.e., the intersections of these edges with the domain boundary. Applying the corresponding stencils [12] to subdivide the structured tetrahedron ($f1$, $f2$, $f3$, p), one obtains three unstructured tetrahedra: ($f1$, p, w, v), ($f3$, p, w, v), and (w, $f3$, $f2$, p). Similarly, for the structured tetrahedron ($f1$, $f2$, $f3$, q), one obtains four unstructured tetrahedra: ($f1$, u, v, w), ($f3$, u, v, w), (w, $f2$, u, $f3$), and (q, $f2$, u, $f3$). Tetrahedra ($f1$, p, w, v) and ($f1$, u, v, w) are classified as located outside the domain, and are not considered in this work. The remaining unstructured tetrahedra, three on one side of the shared face and two on the other side, are classified as lying within the domain and are considered further.

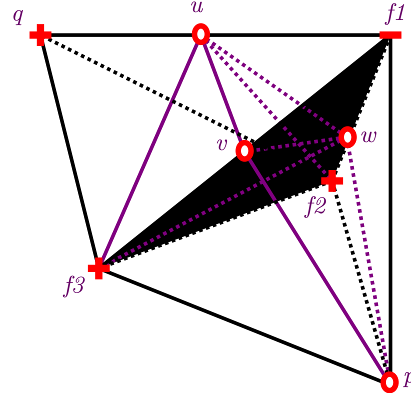

Figure 3: An example of two face-adjacent structured tetrahedra, ($f1$, $f2$, $f3$, p) and ($f1$, $f2$, $f3$, q) shown with black lines, a sign assignment to their vertices, and a subdivision into unstructured tetrahedra shown with purple lines. The shared face ($f1$, $f2$, $f3$) is shaded.

Via an examination of all edges of the unstructured tetrahedra classified as lying inside the domain, it can be concluded that the two sets of edges created inside the two respective structured tetrahedra are consistent

within the shared face. In the rest of this work, formal conditions for such a verification are developed and proven.

Section 2 introduces the methodology used in this work. Section 3 describes the executable functions whose properties are certified. Section 4 presents the formally stated properties of the function that constructs structured tetrahedra and the proof of these properties. Section 5 certifies the function that constructs unstructured tetrahedra, considered separately, in a similar progression. In Section 6 the properties of both functions are proven together, which is necessary since unstructured tetrahedra are constructed out of the structured ones. Section 7 concludes the exposition with the summary of the presented work and expected extensions.

2. METHODOLOGY

The Curry-Howard correspondence is a fundamental concept in proof theory that considers proving to be a kind of programming. Indeed, while a computer program is a sequence of transformations of input data to output data, a proof is a sequence of transformations that takes the input proposition (premise) and converts it to the output proposition (conclusion). In this sense, implementing an algorithm as a function is similar to proving a theorem.

Computer-assisted proving involves a human user who guides the process through formulating intermediate steps and choosing proof techniques. The user interacts with a software program that manages the state of the proof, verifies the validity of the typed commands, and executes those commands.

Proof automation allows for finding proofs in special cases by invoking predefined proof search strategies. Once the user identifies that at a certain stage in the proving process a particular proof goal is suitable for some predefined automatic procedure, that procedure can be invoked in order to reduce the number of manually guided steps.

Purely functional programming is a programming methodology that views the subroutines to be similar to mathematical functions, whose return value depends only on the values of the parameters, and that produce no side effects. This methodology, when enforced in a 'pure' language, often translates to restricting the programming constructs so that there is no memory aliasing via pointers, parameters to functions are passed and returned 'by value', and mutable global values are not allowed. Eliminating side effects and aliasing makes it much easier to argue about the return value of a function, and therefore a pure functional programming language is a suitable choice for a proof assistant. Functional programming is contrasted to imperative programming that changes state with commands, such as, for example, an assignment.

Extraction is a methodology that separates the executable part of a program's specification from its logical part and saves it in a language that supports compilation and deployment.

The Coq formal proof management system [8] was chosen for this development. Coq combines a purely functional programming language called Gallina, which is based on a formal language Calculus of Inductive Constructions, and a vernacular language of commands that allow for stating software specifications and proving mathematical theorems. Some of the Coq's salient features that are relevant to the current exposition are briefly reviewed below. A thorough treatment can be found in relevant texts [7, 16, 17].

Coq's Calculus of Inductive Constructions is a general theory that defines a typed programming language that can serve as constructive foundation of mathematics. Inductive definitions are in the core of Coq's language. For example, a basic number type used in this development, binary integer numbers (Z), are defined inductively in Coq's standard library as follows.

```
Inductive positive : Set :=
  | xI : positive → positive
  | xO : positive → positive
  | xH : positive.
```

```
Inductive Z : Set :=
  | Z0 : Z
  | Zpos : positive → Z
  | Zneg : positive → Z.
```

Type Z can be thought of as a wrapper around the *positive* type, with its constructors corresponding to number 0, some positive number, or some negated positive number, respectively. The *positive* number type is defined via constructor xH, representing binary digit 1 in the most significant position, and two constructors, xO and xI, that add a least significant digit, 0 or 1 respectively, to a given *positive* number. For example, *Zneg (xO (xI xH))* represents -6. The arrow operator $\rightarrow$ is a special case of a universal quantifier and is used in a variety of contexts, including listing function and constructor arguments, as well as logical implication.

Polymorphism is a concept in programming that stands for providing a single interface to elements of different types. Polymorphism is fully supported in Coq by declaring undefined types as variables. In the current development, for example, the concrete representation of the *Vertex* data structure is irrelevant to the proof of unstructured stencil correctness. As a result, in the Coq script it was abstracted as an unspecified type:

```
Variable Vertex : Type.
```

The unstructured tetrahedron data structure is then defined as four *Vertex* data elements appearing as parameters to the tetrahedron constructor *UT*:

```
Inductive UnstructuredTet :=
  UT : Vertex → Vertex →
       Vertex → Vertex → UnstructuredTet.
```

Closure of a function is a function that carries bindings to all the data structures referenced within it [18]. Along with polymorphism, functional closure allows for abstracting implementation details that are not needed for the proof part. One example of closure used in this development is the cut function named *GetSign*, which is bound to the definition of the domain to be meshed in the executable part of the implementation. However, the proof part only needs to know that this function takes a parameter of type *Vertex* and returns its *Sign*:

```
Variable GetSign : Vertex → Sign.
```

Tactics are predefined commands that are executed in the current proof environment to transform and eventually discharge the proof goal. Coq offers a collection of tactics that can accomplish a variety of logical transformations, such as applying a previously proven theorem (`apply`), proof by case enumeration (`destruct`), substitution of previously defined expressions (`unfold`), and many more. Some tactics exhibit a high degree of automation to ease the development effort on the part of the user. For example, one of the tactics used hereby, `eauto`, combines a Prolog-like resolution procedure `auto` with deferred instantiation of existential variables.

Tactical language in Coq is called Ltac. It contains a number of high-level commands that allow for applying tactics in various arrangements, including sequences, loops, and branches. The current development makes use of some of the features of this language, such as repeated application of tactics, catching exceptions, defining automated reusable proof procedures, and matching the shape of the current goal against a given pattern.

3. CERTIFIED EXECUTABLE FUNCTIONS

3.1 Function for Constructing Structured Tetrahedra

The IS algorithm uses structured tetrahedra to fill in the region of space to be meshed in the proximity of the domain boundary. The data structure for structured tetrahedra contains not only their vertex information, but also a sufficient amount of data to query all four of its neighbors adjacent via the faces, along with the vertices of the neighbors that correspond to the vertices of the given tetrahedron.

```
Inductive StructuredTet :=
  ST : Coord3 → Coord3 → Coord3 → Coord3 →
       StructuredTetId × VertexOrder →
       StructuredTetId × VertexOrder →
       StructuredTetId × VertexOrder →
       StructuredTetId × VertexOrder →
       StructuredTet.
```

Here the *Coord3* parameters to the constructor *ST* are the three-dimensional coordinates of its vertices. The *Coord3* data type simply combines three *Z* values corresponding to each of the Euclidean dimensions.

```
Inductive Coord3 :=
  C3 : Z → Z → Z → Coord3.
```

As mentioned above, the data structure for unstructured tetrahedra uses a variable, i.e., unspecified, type *Vertex*. When it is necessary to make the vertex types used in the proofs match, the *Coord3* type is passed as a parameter that instantiates the *Vertex* type.

The type *StructuredTetId* is defined as follows.

```
Inductive StructuredTetId :=
  STId : Coord3 → StructuredTetCase
         → StructuredTetId.
```

The *Coord3* parameter to the constructor *STId* is the three-dimensional coordinate of the reference, or anchor, that defines the position of the tetrahedron.

The type *StructuredTetCase* defines one of 12 possible orientations that are enumerated as constructors of this type:

```
Inductive StructuredTetCase :=
  Tet_I1 | Tet_I2 | Tet_I3 | Tet_I4 |
  Tet_J1 | Tet_J2 | Tet_J3 | Tet_J4 |
  Tet_K1 | Tet_K2 | Tet_K3 | Tet_K4.
```

The *VertexOrder* parameters to the *ST* constructor specify the ordering of the vertices of each neighbor tetrahedron, so that the corresponding vertices can be known (see Section 6). It is a function that returns a permutation of four vertices:

```
Definition VertexOrder : Type :=
  Coord3 → Coord3 → Coord3 → Coord3 →
  (Coord3 × Coord3 × Coord3 × Coord3).
```

Figure 4 illustrates some of these data elements. The cube drawn with black lines has even-coordinate corners, and the one drawn with blue lines has odd-coordinate corners. The corner marked ijk is the reference. The other corner labels indicate the offset from the reference coordinate. The red lines show the edges that are added to form four tetrahedra shown: $(220, 222, 311, 331)$, $(220, 222, 311, 111)$,

(220, 222, 131, 331), (220, 222, 131, 111). These tetrahedra correspond to cases *Tet_K1*, *Tet_K2*, *Tet_K3*, and *Tet_K4*, respectively. Another set of four tetrahedra (not shown) is constructed around edge (202, 222). The third set of four tetrahedra (also not shown) is constructed around edge (022, 222). Twelve tetrahedra correspond to each reference coordinate ijk. The executable function *GetStructuredTet* returns one of these tetrahedra, depending on the case requested.

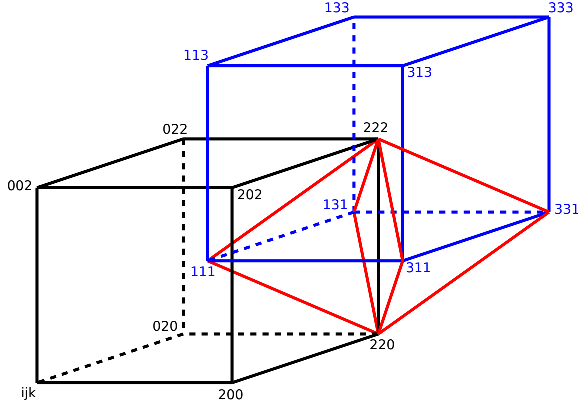

Figure 4: The vertices used by the function that constructs structured tetrahedra.

The structured tetrahedra can be defined by a simple formula that returns their connectivity with respect to the vertices of the BCC grid. This formula will translate and rotate a reference tetrahedron to define all other similar tetrahedra. Such a formula was encoded into an executable function named *GetStructuredTet*.

```
Definition GetStructuredTet (tid : StructuredTetId)
                              : StructuredTet :=
[omitted]
```

3.2 Function for Constructing Unstructured Tetrahedra

The executable function *GetUnstructuredTets* returns a list of tetrahedra that subdivide a given structured tetrahedron. The parameters to this function *v1*, *v2*, *v3*, *v4* are vertices of the given structured tetrahedron. This function enumerates all of the 81 possible combinations of values *GetSign v1*, *GetSign v2*, *GetSign v3*, and *GetSign v4*, and returns a corresponding predefined list of unstructured tetrahedral stencils adopted from the original publication [12].

```
Definition GetUnstructuredTets
             (v1 v2 v3 v4 : Vertex)
             : list UnstructuredTet :=   [omitted]
```

4. CERTIFICATE FOR STRUCTURED TETRAHEDRA

4.1 Specification

The specification for the correctness of the structured tetrahedron returned by function *GetStructuredTet* states the following requirement. Let *v1*, *v2*, *v3*, *v4* be the vertices of this tetrahedron, and *nei_tid* and *nei_order* be the identificator and the order of vertices of its i-th neighbor ($i = 1, 2, 3, 4$). A convention is used that the i-th neighbor is the one that is across from the i-th vertex. Then, if the same function *GetStructuredTet* is called for *nei_tid*, and the returned vertices *u1*, *u2*, *u3*, *u4* are permuted by function *nei_order*, then the vertices with the corresponding positions are equal, except for the vertices in position i that are not equal. The `Section` mechanism of Coq allows for moving the frequently used parameters of the constructs defined within this section to its `Variables` clause, thus improving readability.

```
Section GetStructuredTet_Spec.
  Variables (v1 v2 v3 v4 : Coord3)
            (nei_tid : StructuredTetId)
            (nei_order : VertexOrder).
  Definition U :=
    match GetStructuredTet nei_tid with
      ST u1 u2 u3 u4 _ _ _ _ ⇒
        nei_order u1 u2 u3 u4
    end.
  Definition Face1Correct :=
    match U with (u1, u2, u3, u4) ⇒
      ~(v1 = u1) ∧ (v2 = u2) ∧
      (v3 = u3) ∧ (v4 = u4)
    end.
  Definition Face2Correct :=
    match U with (u1, u2, u3, u4) ⇒
      (v1 = u1) ∧ ~(v2 = u2) ∧
      (v3 = u3) ∧ (v4 = u4)
    end.
  [omitted]
End GetStructuredTet_Spec.
```

4.2 Proof

The following theorem proves the specification above.

```
Theorem FacesCorrect :
  ∀ tid : StructuredTetId,
  match GetStructuredTet tid with
    ST v1 v2 v3 v4 (tid1, o1) (tid2, o2)
                   (tid3, o3) (tid4, o4) ⇒
      (Face1Correct v1 v2 v3 v4 tid1 o1) ∧
      (Face2Correct v1 v2 v3 v4 tid2 o2) ∧
      (Face3Correct v1 v2 v3 v4 tid3 o3) ∧
```

```
    (Face4Correct v1 v2 v3 v4 tid4 o4)
  end.
Proof.    [omitted]  Qed.
```

5. CERTIFICATE FOR UNSTRUCTURED TETRAHEDRA

5.1 Specification

Given two edges, each belonging to the stencils on either side of the shared face, these edges are considered compatible if they do not intersect in their interiors. A set of conditions is developed below that allow for verifying this requirement.

5.1.1 Barycentric Coordinate System

Let $v_1, \ldots, v_n$ be the vertices of a simplex in Euclidean space $\mathbb{R}^3$, given as triples of coordinates. For the present exposition, the relevant values of n are 3 and 4, corresponding to a triangle or a tetrahedron, respectively. For some point $u \in \mathbb{R}^3$, the real numbers $a_1, \ldots, a_n$, not all equal to zero, such that $(a_1 + \cdots + a_n)u = a_1 v_1 + \cdots + a_n v_n$, are called barycentric coordinates of u with respect to the simplex with vertices $v_1, \ldots, v_n$. Barycentric coordinates with $a_1 + \cdots + a_n = 1$ will be used below. For the proofs presented, only three values of a barycentric coordinate are relevant: zero, some unknown value strictly between zero and one, and one. These are defined symbolically by the following respective constructors of type *BCoord*:

```
Inductive BCoord := Zero | Interior | One.
```

5.1.2 Barycentric Coordinates Within a Triangle

Consider the barycentric coordinate system defined by the shared face (*f1*, *f2*, *f3*). Then the following proposition *FCoord* lists the barycentric coordinates of vertices *f1*, *f2*, *f3*, as well as of the vertices computed on the edges of this face by function *GetIntersection*:

```
Variables f1 f2 f3 p q : Vertex.
Inductive FCoord : Vertex → BCoord →
                    BCoord → BCoord → Prop :=
  | FC_1 : FCoord f1 One Zero Zero
  | FC_2 : FCoord f2 Zero One Zero
  | FC_3 : FCoord f3 Zero Zero One
  | FC_4 : FCoord (GetIntersection f1 f2)
                  Interior Interior Zero
  | FC_5 : FCoord (GetIntersection f2 f1)
                  Interior Interior Zero
  | FC_6 : FCoord (GetIntersection f1 f3)
                  Interior Zero Interior
  | FC_7 : FCoord (GetIntersection f3 f1)
                  Interior Zero Interior
  | FC_8 : FCoord (GetIntersection f2 f3)
                  Zero Interior Interior
  | FC_9 : FCoord (GetIntersection f3 f2)
                  Zero Interior Interior.
```

5.1.3 Barycentric Coordinates Within a Tetrahedron

For the tetrahedron (*f1*, *f2*, *f3*, *p*), the barycentric coordinate system is defined similarly. For the proof that follows, however, the full coordinates of the vertices are not needed. The only needed piece of information is which vertices lie off the shared face (*f1*, *f2*, *f3*), as evidenced by their non-zero last barycentric coordinate (i.e., the one corresponding to vertex *p*). The proposition below makes this information available.

```
Inductive PCoord : Vertex → Prop :=
  | PC_1 : PCoord p
  | PC_2 : PCoord (GetIntersection p f1)
  | PC_3 : PCoord (GetIntersection f1 p)
  | PC_4 : PCoord (GetIntersection p f2)
  | PC_5 : PCoord (GetIntersection f2 p)
  | PC_6 : PCoord (GetIntersection p f3)
  | PC_7 : PCoord (GetIntersection f3 p).
```

The predicate *QCoord* is defined similarly to *PCoord*, only with respect to vertex *q* of tetrahedron (*f1*, *f2*, *f3*, *q*).

5.1.4 Barycentric Non-Intersection in the Shared Face

Given a point *p* and a segment (*q1*, *q2*), both lying in face (*f1*, *f2*, *f3*), a proposition can be stated that guarantees *p* not being in the interior of (*q1*, *q2*). Let the barycentric coordinates of points *q1*, *q2*, *p* with respect to vertex *f1* be *a*, *b*, *c*, respectively, see Figure 5. Then, if *c* is above or below *a* and *b*, as specified precisely by the propositions that follow, this requirement can be formalized.

Proposition *Below* enumerates the cases (i.e., combinations of values of its parameters) that assure the required property of segment-point non-intersection, where the first parameter is the coordinate of the point, while the second and the third parameters are the coordinates of the segment's vertices. Care is taken not to make conclusions based on comparing two *Interior* coordinates, since their specific values are unknown before they are computed.

```
Inductive Below : BCoord → BCoord →
                  BCoord → Prop :=
  | B_1 : Below Zero Zero Interior
  | B_2 : Below Zero Zero One
  | B_3 : Below Zero Interior Interior
```

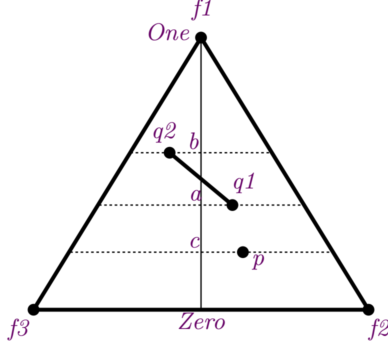

Figure 5: Point *p* with barycentric *f1*-coordinate *c* not lying in the interior of segment (*q1*, *q2*) whose vertices have barycentric *f1*-coordinates *a* and *b*.

```
  | B_4 : Below Zero Interior One.
```

Proposition *Above* is stated similarly.

```
Inductive Above : BCoord → BCoord →
                   BCoord → Prop :=
  | A_1 : Above One Zero Interior
  | A_2 : Above One Zero One
  | A_3 : Above One Interior Interior
  | A_4 : Above One Interior One.
```

In order to allow the vertices of the segment to appear in an arbitrary order, symmetric versions of propositions *Below* and *Above* are defined. They simply refer to these propositions with both orderings of parameters.

```
Section Below_Above_Symm.
  Variables c a b : BCoord.
  Inductive BelowSymm : Prop :=
  | BS_1 : Below c a b → BelowSymm
  | BS_2 : Below c b a → BelowSymm.
  Inductive AboveSymm : Prop :=
  | AS_1 : Above c a b → AboveSymm
  | AS_2 : Above c b a → AboveSymm.
End Below_Above_Symm.
```

The proposition *SPNI1D* below states that *BCoord* value *c* is either above or below values *a* and *b* (regardless of the ordering of *a* and *b*), which guarantees that a point with coordinate *c* does not lie in the interior of the segment whose vertices have corresponding coordinates *a* and *b*.

```
Section Segment_Point_Not_Intersecting_1D.
  Variables c a b : BCoord.
  Inductive SPNI1D : Prop :=
  | SPNI1D_1 : BelowSymm c a b → SPNI1D
  | SPNI1D_2 : AboveSymm c a b → SPNI1D.
End Segment_Point_Not_Intersecting_1D.
```

If the non-intersection condition *SPNI1D* holds with respect to at least one barycentric coordinate, then the vertex with barycentric coordinates (*c1*, *c2*, *c3*) does not intersect the interior of the segment whose vertices have coordinates (*a1*, *a2*, *a3*) and (*b1*, *b2*, *b3*):

```
Section Segment_Point_Not_Intersecting_3D.
  Variables a1 a2 a3 b1 b2 b3 c1 c2 c3 : BCoord.
  Inductive SPNI3D : Prop :=
  | SPNI3D_1 : SPNI1D c1 a1 b1 → SPNI3D
  | SPNI3D_2 : SPNI1D c2 a2 b2 → SPNI3D
  | SPNI3D_3 : SPNI1D c3 a3 b3 → SPNI3D.
End Segment_Point_Not_Intersecting_3D.
```

A similar approach is used in the case of two segments lying in face (*f1*, *f2*, *f3*), see Figure 6.

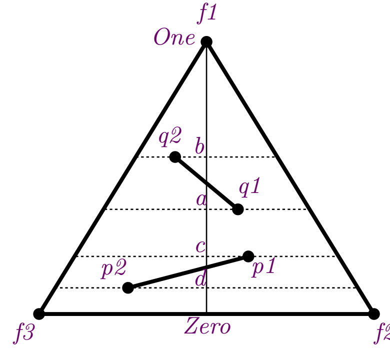

Figure 6: Segment (*p1*, *p2*) whose vertices have barycentric *f1*-coordinates *c* and *d* and segment (*q1*, *q2*) whose vertices have barycentric *f1*-coordinates *a* and *b* not intersecting in their interiors.

First, a proposition is stated that establishes a one-dimensional barycentric condition of their non-intersection in the interiors. Given that the coordinates of the first segment's vertices are *a* and *b*, and the coordinates of the second segment's vertices are *c* and *d*, the condition requires that either both vertices of the first segment are below the vertices of the second segment or vice versa:

```
Section Segment_Segment_Not_Intersecting_1D.
  Variables a b c d : BCoord.
  Inductive SSNI1D : Prop :=
  | SSNI1D_1 : BelowSymm a c d →
               BelowSymm b c d → SSNI1D
  | SSNI1D_2 : BelowSymm c a b →
               BelowSymm d a b → SSNI1D.
End Segment_Segment_Not_Intersecting_1D.
```

Then a proposition is stated requiring that the one-dimensional condition holds with respect to at least one of three barycentric dimensions.

```
Section Segment_Segment_Not_Intersecting_3D.
  Variables a1 a2 a3 b1 b2 b3
            c1 c2 c3 d1 d2 d3 : BCoord.
  Inductive SSNI3D : Prop :=
  | SSNI3D_1 : SSNI1D a1 b1 c1 d1 → SSNI3D
```

```
  | SSNI3D_2 : SSNI1D a2 b2 c2 d2 → SSNI3D
  | SSNI3D_3 : SSNI1D a3 b3 c3 d3 → SSNI3D.
End Segment_Segment_Not_Intersecting_3D.
```

5.1.5 Barycentric Non-Intersection Within Adjacent Tetrahedra

In this section, a condition is established that formally defines a sufficient condition for two edges, each incident on the irregular tetrahedra on either side of the shared face, to be compatible, i.e., not intersecting in their interiors. This condition is given by the proposition *TwoEdgesCompatible* stated below. Two edges, (*p1*, *p2*) and (*q1*, *q2*), are considered compatible if at least one of the following conditions is satisfied:

- Vertex *p1* is in a special position (proposition *SpecialVertex* described further below) with respect to edge (*q1*, *q2*) and vertex *p2* is not in the interior of edge (*q1*, *q2*) (proposition *NotIn* described further below).
- The condition above holds if vertices *p1* and *p2* are interchanged.
- The two conditions above hold if edges (*p1*, *p2*) and (*q1*, *q2*) are interchanged.
- If both edges (*p1*, *p2*) and (*q1*, *q2*) lie in the shared face (*f1*, *f2*, *f3*), then they do not intersect in their interiors as defined by proposition *SSNI3D* above.

```
Section Two_Edges_Compatible.
  Variables p1 p2 q1 q2 : Vertex.
  Inductive TwoEdgesCompatible : Prop :=
  | EC_1 : SpecialVertex p1 q1 q2 PCoord →
            NotIn p2 q1 q2 PCoord QCoord →
            TwoEdgesCompatible
  | EC_2 : SpecialVertex p2 q1 q2 PCoord →
            NotIn p1 q1 q2 PCoord QCoord →
            TwoEdgesCompatible
  | EC_3 : SpecialVertex q1 p1 p2 QCoord →
            NotIn q2 p1 p2 QCoord PCoord →
            TwoEdgesCompatible
  | EC_4 : SpecialVertex q2 p1 p2 QCoord →
            NotIn q1 p1 p2 QCoord PCoord →
            TwoEdgesCompatible
  | EC_5 : ∀ a1 a2 a3 b1 b2 b3
             c1 c2 c3 d1 d2 d3 : BCoord,
            FCoord p1 a1 a2 a3 →
            FCoord p2 b1 b2 b3 →
            FCoord q1 c1 c2 c3 →
            FCoord q2 d1 d2 d3 →
            SSNI3D a1 a2 a3 b1 b2 b3
                   c1 c2 c3 d1 d2 d3 →
            TwoEdgesCompatible.
End Two_Edges_Compatible.
```

The special status of vertex *v* with respect to edge (*u1*, *u2*) here means that *v* is equal to one of the vertices *u1* or *u2*, or it is off the shared face as defined by the corresponding parameter proposition *V* (equal to *PCoord* or *QCoord* depending on the branch of *TwoEdgesCompatible*).

```
Section Special_Vertex.
  Variables (v u1 u2 : Vertex)
            (V : Vertex → Prop).
  Inductive SpecialVertex : Prop :=
  | SV_1 : v = u1 → SpecialVertex
  | SV_2 : v = u2 → SpecialVertex
  | SV_3 : V v → SpecialVertex.
End Special_Vertex.
```

Vertex *v* being *NotIn* edge (*u1*, *u2*) holds if at least one of the following conditions is satisfied:

- *v* is a special vertex with respect to *u1*, *u2* as defined above.
- At least one of *u1*, *u2* is off the shared face as defined by the appropriate proposition *PCoord* or *QCoord*.
- All three vertices *v*, *u1*, *u2* lie in the shared face and *v* is not in the interior of edge *u1*, *u2* as defined by proposition *SPNI3D*.

```
Section Not_In.
  Variables (v u1 u2 : Vertex)
            (V U : Vertex → Prop).
  Inductive NotIn : Prop :=
  | NI_1 : SpecialVertex v u1 u2 V → NotIn
  | NI_2 : U u1 → NotIn
  | NI_3 : U u2 → NotIn
  | NI_4 : ∀ a1 a2 a3 b1 b2 b3 c1 c2 c3 : BCoord,
           FCoord u1 a1 a2 a3 →
           FCoord u2 b1 b2 b3 →
           FCoord v c1 c2 c3 →
           SPNI3D a1 a2 a3 b1 b2 b3 c1 c2 c3 →
           NotIn.
End Not_In.
```

Proposition *AllEdgesCompatible* below makes use of *TwoEdgesCompatible* by passing to it every pair of edges, each pair consisting of edges from both sides of the shared face.

```
Definition AllEdgesCompatible
           (u1 u2 u3 u4 v1 v2 v3 v4 : Vertex)
           : Prop :=
  let T := GetUnstructuredTets Vertex
             GetSign GetIntersection in
  ∀ (t1 t2 : UnstructuredTet Vertex),
  List.In t1 (T u1 u2 u3 u4) →
  List.In t2 (T v1 v2 v3 v4) →
```

```
Forall EdgePairCompatible
        (list_prod (Edges t1) (Edges t2)).
```

Given two regular tetrahedra, one defined by vertices *u1*, *u2*, *u3*, *u4*, and the other by *v1*, *v2*, *v3*, *v4*, function *GetUnstructuredTets* is called for each of these tetrahedra. This function returns a list of irregular tetrahedra for each regular tetrahedron. Let *t1* be an arbitrary tetrahedron in the list returned for *u1*, *u2*, *u3*, *u4*. Let *t2* be an arbitrary tetrahedron in the list returned for *v1*, *v2*, *v3*, *v4*. Six edges of *t1* and six edges of *t2* are collected into two respective lists using function *Edges* below:

```
Definition Edges (t : UnstructuredTet Vertex)
            : list Edge :=
  match t with UT _ u v w r ⇒
    [(u, v); (u, w); (u, r); (v, w); (v, r); (w, r)]
  end.
```

The standard library function *list_prod* returns a cartesian product of two lists, which are *Edges t1* and *Edges t2* in this development. Finally, the standard library proposition *Forall* asserts that a proposition supplied as its first parameter (*EdgePairCompatible*) holds for all elements of the list supplied as its second parameter. *EdgePairCompatible* simply unpacks the vertices of both edges and passes them to *TwoEdgesCompatible*:

```
Definition EdgePairCompatible ee :=
  let '((p1, p2), (q1, q2)) := ee in
  TwoEdgesCompatible p1 p2 q1 q2.
```

5.2 Proof

The following lemma proves that the proposition *AllEdgesCompatible* holds for six arrangements of five vertices *f1*, *f2*, *f3* *p*, *q* that define two tetrahedra with shared face (*f1*, *f2*, *f3*).

```
  Lemma CompatibilityOrders :
    ∀ Vertex GetSign GetIntersection f1 f2 f3 p q,
    let P := AllEdgesCompatible Vertex GetSign
                 GetIntersection f1 f2 f3 p q in
    P p f1 f2 f3 q f1 f2 f3 ∧
    P p f1 f2 f3 f1 q f3 f2 ∧
    P f1 p f2 f3 q f1 f3 f2 ∧
    P f1 p f2 f3 f1 q f2 f3 ∧
    P f1 f2 p f3 f1 f2 q f3 ∧
    P f1 f2 f3 p f1 f2 f3 q.
  Proof.    [omitted]   Qed.
```

6. CERTIFICATE FOR STRUCTURED AND UNSTRUCTURED TETRAHEDRA

In this section, the correctness of the two functions, one returning structured and the other returning unstructured tetrahedra, is proven when they are used together. Function *GetStructuredTet* returns a structured tetrahedron which is then passed to *GetUnstructuredTets*.

```
Theorem Face1Compatible :
  ∀ GetSign GetIntersection tid,
  match GetStructuredTet tid with
    ST v1 v2 v3 v4 (tid1, order1) _ _ _ ⇒
      match GetStructuredTet tid1 with
        ST u1 u2 u3 u4 _ _ _ _ ⇒
          let '(w1, w2, w3, w4) :=
            order1 u1 u2 u3 u4 in
          AllEdgesCompatible Coord3 GetSign
            GetIntersection w2 w3 w4 v1 w1
            v1 v2 v3 v4 u1 u2 u3 u4
      end
  end.
Proof.    [omitted]   Qed.
```

The statement of this theorem is somewhat similar to that of theorem *FacesCorrect*. Both theorems examine the result of two calls to function *GetStructuredTet* with respect to the face shared by the two returned structured tetrahedra. Theorem *FacesCorrect* proves that the shared face is identified correctly. Theorem *Face1Compatible*, on the other hand, proves that two sets of unstructured tetrahedra, each computed via function *GetUnstructuredTets* within the definition of proposition *AllEdgesCompatible*, satisfy this proposition. The proof of theorem *Face1Compatible* makes use of lemma *CompatibilityOrders* that requires one of six specific orderings of vertices of the two structured tetrahedra. The orderings of vertices returned by the calls to *GetStructuredTet* satisfy this requirement. Another three theorems are proven, *Face2Compatible*, *Face3Compatible* and *Face4Compatible*, that are identical to *Face1Compatible* except for the faces that are examined.

7. DISCUSSION AND CONCLUSIONS

Currently, major software projects consist of millions of lines of code, multiple subsystems, and are managed by large teams of developers (who tend to move between positions). With this level of complexity, formulating and maintaining the specifications of such systems, and ensuring that the implementation meets these specifications become a major challenge. The traditional approach to software specifications consists in writing textual descriptions in English (or other natural language) with a mix of mathematical statements, and maintaining them as code comments and/or as separate documents. There are two problem with such specifications. The first is that they tend to be high-level and not necessarily reflective of the rich set of behaviors the code can exhibit. The second problem is that they are disconnected from the code, in the sense

that updates to the code and to these specifications may be performed separately and not necessarily consistently with each other. The certified software development process illustrated in this paper solves both of these problems by virtue of the specifications referring to the actual implementation. It solves the first problem by using specifications stated in a precise logical language, and having these statements automatically checked for exhaustive coverage of all possible cases. It solves the second problem by having the specification directly reference the implementation. Appel et al. [9] coined the term *deep specification* to refer to specifications that are simultaneously *rich* (sufficiently detailed), *two-sided* (both implementable and useful), *formal* (stated in a formal language which supports automated tools), and *live* (connected to implementation). A number of model checking and design specification approaches based on formal languages like Alloy [5], AADL [19], VDM [20], and Z [21], operate on a high level of abstraction that is not connected with the implementation. An approach of annotating the routines with pre- and post-conditions, such as Design by Contract [22], relies on the developer's current understanding of these conditions and is also not live.

In addition to providing live specifications of software, the presented certification approach also guarantees that the proven properties hold for all input parameters that can be passed to the routines. In other words, it *by design* eliminates the need to test the resulting software for these properties. The caveat, however, is that the formally stated specifications have to correctly represent the expected properties.

Mesh generation is one of the application domains that is likely to benefit from the use of formal methods in general and of certified software development in particular. One reason behind this expectation is high complexity of mesh generation algorithms that need to balance multiple, often contradictory, requirements of element shape, boundary fidelity, mesh grading, number of elements, software running time and memory use, and others. These requirements can be thought of as a multidimensional design space, where each requirement represents one dimension of this space. Being able to *a priori* formally guarantee, or certify, certain properties of an algorithm's implementation allows for the reduction of the dimensionality of the remaining design space. An example of this phenomenon from the current work is the unfolding of the tetrahedral stencils: because the unfolded stencils have been automatically checked for consistency during a single certification event, the Parity Rule [12] does not need to be enforced at runtime during every execution of the software. Another reason behind the expectation of the usefulness of formal methods in mesh generation is the combinatorial nature of mesh stencils, or elements, which is suitable for automated enumeration and analysis of their possible arrangements.

The efficiency of the proof execution does not influence the performance of the final software product. Indeed, the proof is run at design time and then separated from the executable part by the process of extraction. This executable part, on the other hand, is the component which gets deployed and whose performance contributes to the efficiency of the resulting product. The main requirement on the proof efficiency is that it does not unreasonably slow down the design effort. The experience of the author of this work suggests that the proof design effort is dominated by the time spent on elucidating the properties that need to be proven, formalizing them, and discovering the appropriate proof strategies. The proof execution part appeared minor relative to these time investments. It was also noticed that the same result can be proven with different strategies that sometimes vary significantly in performance. The accompanying certificate [14] was run on a MacBook Pro equipped with an 8-core Intel Core i9 @ 2.4 GHz processor and 32 GB of RAM memory. In the serial mode the proof completed in 46 minutes. Virtually all of this time was taken by the proof of lemma *CompatibilityOrders*. Experimentation with Coq's parallel proof modes revealed that this time can be reduced to 31 minutes with two proof threads. Increasing the number of proof threads to six, which is the number of independent proof goals in this lemma, did not yield further speedup.

The present work lays out a methodological foundation and a case study for developing certified mesh generation software. It demonstrates the use of the Calculus of Inductive Constructions for defining executable functions, stating their properties, and proving these properties via a direct examination of these functions (the property of liveness). These functions were extracted into OCaml code, supplemented with other OCaml functions needed for a working computer program, compiled, and executed. The evaluation of the performance of this program, as well as of the choices of the data structures, is out of the scope of this exposition and will be presented elsewhere.

It is anticipated that this work will be extended in the following directions. The first direction is the live proofs of the angle bounds reported in the original IS presentation [12]. The second direction is the addition of other shapes of structured tetrahedra that allow for more flexibility in filling in the octree leaves, leading to fewer resulting tetrahedra, as also originally reported [12]. The third direction is the study of stencils for multi-material interfaces, which can potentially be informed by previous work [23].

References

[1] Newcombe C., Rath T., Zhang F., Munteanu B., Brooker M., Deardeuff M. "How Amazon Web Services Uses Formal Methods." *Communications of the ACM*, vol. 58, no. 4, 66–73, Mar. 2015

[2] Lamport L. "TLA+ Hyperbook." `http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html`, 2015

[3] Eriksen M. "Functional at Scale." *Communications of the ACM*, vol. 59, no. 12, 50–55, Dec. 2016

[4] Avigad J., Harrison J. "Formally Verified Mathematics." *Communications of the ACM*, vol. 57, no. 4, 66–75, Apr. 2014

[5] Jackson D. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edn., 2012

[6] Chlipala A. "Ur/Web: A Simple Model for Programming the Web." *Communications of the ACM*, vol. 59, no. 8, 93–100, Jul. 2016

[7] Bertot Y., Castéran P. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004

[8] INRIA France. "The Coq Proof Assistant, version 8.9.1." `http://coq.inria.fr`

[9] Appel A.W., Beringer L., Chlipala A., Pierce B.C., Shao Z., Weirich S., Zdancewic S. "Position Paper: The Science of Deep Specification." *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, 20160331, 2017

[10] Chernikov A., Xu J. "A computer-assisted proof of correctness of a marching cubes algorithm." *International Meshing Roundtable*, pp. 505–523. Springer, Orlando, FL, October 2013

[11] Lorensen W.E., Cline H.E. "Marching Cubes: A high resolution 3D surface construction algorithm." *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, 163–169, Aug. 1987

[12] Labelle F., Shewchuk J.R. "Isosurface Stuffing: Fast Tetrahedral Meshes with Good Dihedral Angles." *ACM Transactions on Graphics*, vol. 26, no. 3, 57.1 – 57.10, 2007

[13] Dufourd J.F., Bertot Y. "Formal Study of Plane Delaunay Triangulation." M. Kaufmann, L.C. Paulson, editors, *Interactive Theorem Proving, Lecture Notes in Computer Science*, vol. 6172, pp. 211–226. Springer, 2010

[14] Chernikov A. "Coq Script: Certified Functions for Mesh Generation." `https://github.com/anchernikov/IMR2019`, 2019

[15] Talos I., Jakab M., Kikinis R., Shenton M. "SPL-PNL Brain Atlas." `http://www.spl.harvard.edu/publications/item/view/1265`, Mar. 2008

[16] Pierce B.C., et al. *Software Foundations*. 2019. `https://softwarefoundations.cis.upenn.edu`

[17] Chlipala A. *Certified Programming with Dependent Types*. 2019. `http://adam.chlipala.net/cpdt`

[18] Ford N. *Functional Thinking: Paradigm Over Syntax*. O'Reilly Media, 2014

[19] Feiler P.H., Gluch D.P. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012

[20] Larsen P.G. "Ten Years of Historical Development "Bootstrapping" VDMTools." *Journal of Universal Computer Science*, vol. 7, no. 8, 692–709, 2001

[21] Spivey M.J. *The Z Notation: A Reference Manual*. Prentice Hall, 1992

[22] Meyer B. "Applying "Design by Contract"." *IEEE Computer*, vol. 25, no. 10, 40–51, 1992

[23] Bronson J.R., Levine J.A., Whitaker R.T. "Lattice Cleaving: Conforming Tetrahedral Meshes of Multimaterial Domains with Bounded Quality." *Proceedings of the 21st International Meshing Roundtable*, pp. 191–209. Springer, San Jose, CA, 2013