ELSEVIER

Advances in Engineering Software xx (2004) 1–15

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

Semi-formal design of reliable mesh generation systems

A.H. ElSheikh[a], S. Smith[b], S.E. Chidiac[c,*]

[a]Department of Civil Engineering, McMaster University, Hamilton, Ont., Canada, L8S 4L7
[b]Department of Computing and Software, McMaster University, Hamilton, Ont., Canada, L8S 4L7
[c]Department of Civil Engineering, McMaster University, Hamilton, Ont., Canada, L8S 4L7

Received 28 October 2003; revised 20 May 2004; accepted 10 June 2004

Abstract

A reliable mesh generation infrastructure is designed based on software engineering principles. Formal methods, software design documents and clear modular decomposition criteria are introduced to improve the quality of mesh generation software. The design document for a simple 2D mesh generation data structure is presented using a semi-formal specification. The proposed semi-formal documentation system avoids any ambiguity during the software design process and will help in driving the software test cases. Using the proposed software, design techniques result in a consistent software design that is easy to extend and modify.
© 2004 Published by Elsevier Ltd.

Keywords: Mesh generation; Software engineering; Modular decomposition; Formal methods; Software quality

1. Introduction

Mesh generation is an essential component in many numerical methods used for physical simulation. The accuracy of the finite element and the finite volume methods heavily depend on the mesh to be used for the discretization process. The requirements of adaptive numerical methods where mesh modification is needed to increase the accuracy of the solution increases the design complexity of the mesh generation toolboxes. Attempts have been made to improve the design of mesh generators [3,16]. These attempts have identified many of the mesh generation software requirements [3]. One of the major drawbacks of these attempts is a high dependency on a specific implementation language, which was C++ in both cases. In the current practice object oriented methods are usually confused with software engineering principles. It should be clarified that object oriented languages facilitate and encourage many software engineering principles such as data abstraction, information hiding, encapsulation, module generalization and template implementation, but all these concept can be implemented by any well-designed imperative language in combination with disciplined programming practices.

Instead of software specification many programs substitue informal descriptions and comments throughout the program code. Visual specification languages like UML [10] can be used effectively for a pictorial representation of architectural concepts, but these cannot be used to specify mathematical operations or pre- and post-conditions and they lack a mathematically rigorous semantics [6]. This informal way of designing and specifying software poses hardships on all the stages of the software development process that follow. The ability to verify and validate the correctness of the system is missing because of the absence of a reference that specifies the correct software behavior. As a consequence of the above point, software reuse, maintainability and extendability are extremely difficult within the current mesh software development practices.

Recent work [5] suggests that software engineering principles can help with these problems. Whereas Ref. [5] takes a breadth approach and considers several stages of the software cycle, the current work will take a more specific perspective by incorporating three major ideas to improve the quality of mesh generation software. These ideas are formal methods, software design documents and a clear modular decomposition criteria for mesh generation software systems. Formal methods are collections of

* Corresponding author.

0965-9978/$ - see front matter © 2004 Published by Elsevier Ltd.
doi:10.1016/j.advengsoft.2004.06.012

mathematical notations and techniques for describing and analyzing systems [14]. This paper will embrace formal methods as a particular method for increasing software quality and will focus on the process of describing software systems with formal methods. The process of analyzing the software description can be done through the verification process, which can be done deductively, or by testing. Some tools like PVS can be used in the verification [17], but they are hard to use and limited to simple data structures.

Software design documents are a set of separate documents targeting different stages of the software design process. In many cases, these documents are ambiguous or not complete. Specification documents are important for communicating ideas between different parts of the software development team. In this paper, we suggest using a semi-formal language for documenting mesh generation software design so that we can be as specific as necessary.

The ultimate goal of any mesh generation software is to be correct and this correctness is based on analyzing the relation between the computer program p and the specification s. This relation can appear in three different classes of problems. If we are given the specification s and a program p that satisfies these specification needed, then we are dealing with a design problem. If both the specification s and program p are given we can check whether p satisfies s and hence we are dealing with a validation problem. The third case is when we have a program p and we want to extract the specification s of this program. In this case, we are dealing with a reverse engineering problem. The previous three problems may initially look different, but they have many overlapping issues, such as the syntax and semantics to be used and the underlying mathematics of the specification.

The last idea suggested for increasing the quality of the meshing software is the use of a clear modular decomposition criteria. Modular decomposition is the process of dividing a big job into a set of jobs which are small, easy to understand and as independent as possible. The decomposition process may be based on different goals such as, design generality, simplicity, efficiency or the flexibility for certain changes. Identifying the criteria for decomposition rules will result in software code that is consistent with the targeted design.

This paper starts with defining the notation used to specify software components semi-formally. A discussion of the theoretical bases of modelling software systems as state machine is also presented. A simple way for specifying the Module State Machines (MSMs) by both defining the Module Interface Specification (MIS) and the semantics of the transition functions is outlined. The basic rules of modular structure design of software system will be discussed. Finally, a sample design specification document of a 2D unstructured mesh generation data structure followed by the specifications of the Delaunay insertion algorithm is presented. This algorithm will show how to apply the formal methods to this class of problem.

2. Notations

The semi-formal language used throughout this paper is based on simple set notations and first-order logic. This language has atomic types *int*, *bool*, *char*, *string* and *real*. These atomic types can be used in tuples or collections. The syntax used for tuples is ($Type1$, $Type2$,…,$TypeN$) with a semantics of N elements of types $TypeI$ where $I=1, 2,\dots,N$. Internal fields in the tuple can be referenced using the dot notation. For example, if TB is a tuple (var_1: $Type1$, var_2: $Type2$) then the first field can be referenced as $TB.var_1$. Collections of elements are stored in containers, which may be ordered or un-ordered collections with unlimited size. For unordered collections without duplicate elements, the syntax $(Type1)_{set}$ is used for describing a set of $Type1$ which has no limit on the size, in the same sense as an abstract datatype; that is, a set is a mathematical notion independent of any concrete implementation.

One way for defining sets is by constructors that select all the elements of some type that satisfy a given predicate. For example, $S=\{x\colon int|ODD(x)\}$ is a predicate specification for set S of all odd integers [15]. Ordered collections are described by sequences with $(Type2)_{seq}$ as a syntax for sequences. Sequences are unlimited in size in the same sense as an abstract datatype. Sequences are indexed using conventional array notation: $s=\langle s[0],\ s[1],\dots,s[n-1]\rangle$. Adding elements to sequences is done be using the appending symbol $\|$. Concatenation of two sequences is done using the same symbol. The concatenation can be done from the head or the tail of the sequence only. To specify the size of a sequence, or set, a norm notation is used; for example, $|s|$ is used to show the size of the collection s.

Simple prepositional logic operators will be used throughout our specification language. Propositional variables with binary value of TRUE or FALSE are used, along with simple formula including the boolean operators $\wedge$, $\vee$ and $\neg$. First order quantifiers like $\forall$ and $\exists$ will be used as prefix for formulas especially when dealing with sets and sequences. A comprehensive introduction to prepositional logic can be found in Ref. [8].

3. Modelling of meshing software

Modelling is the process of abstraction of the system while preserving a limited number of original details. In this process, the main properties of the system are highlighted to allow better management of complex systems. Modelling software system relies on the concept of state. The state of software can be abstracted into a set of state variables. The size of this set depends on the level of refinement of the model. These state variables capture information about certain steps in the executions path of the software. This information may be the size and content of some data structure or may be a flag for some condition. The set of state variables can be called initial, intermediate or final state depending on the point of program execution. The relation between the initial state and the final state is of great importance because it can be used in defining both pre-conditions and post-conditions, which are widely used in the verification process.

A software system is composed of smaller pieces of software called Modules. A module is a self-contained work assignment for a programmer or programming team [12]. A module can be modelled mathematically as a state machine. A simple form of the formalism for this model as a state machine is a tuple (S, s_0, I, O, T, E) [7] where S is the set of states and s_0 is the initial state and $s_o \in S$, I is a set of inputs, O is a set of outputs and T is the transition function T: $S^*I \rightarrow S$ and E is the output function (E: $S^*I \rightarrow O$). The domain of both T and E are S^*I where the * denotes the Cartesian product. This way of description as MSMs [7] can provide an easy mathematical basis for specifying software modules. Comprehension of the state machine in relational form may be tedious and time consuming. However, a simple method for a complete description of the MSM can be done by listing the state variables and specifying the interface of access functions which change the state variables and produce outputs. The state variables definition is done by listing the name and type of each state variable. Access function are defined by listing the name of the function and types of the input and return values of the function. A mathematical description of the semantics of each function also needs to be given. This method of specifying modules is referred to as a MIS.

4. The modular structure

The first step of designing any software system is to decompose the software into a set of simpler problems through what is called the modular decomposition process. The five goals of modular decomposition as highlighted by Parnas [12] are:

(1) Each module should have a simple structure that can be understood by any programmer who is not a member of the development team.
(2) Each module should be self-contained and the coupling between modules should be minimized. This allows changing the implementation of one module without complete knowledge of other modules and without affecting the behavior of other modules.
(3) The module interface should be flexible so that it can accommodate internal changes of the module without any external changes. Interface changes are avoided because this would export the effect of internal module changes into other modules.
(4) Ideally major changes in the software should be done as a set of independent changes to individual modules.
(5) Understanding the functionality of each module should be possible without knowing the internal details of the module design.

The adopted module decomposition criteria are based on the principles of information hiding, design for change and stepwise refinement. According to information hiding principle, details that are likely to change should be the secrets of separate modules [13].

These ideas of Modular Decomposition can be applied easily to the data structures used in the mesh generator. Any data structure which is expected to change under any circumstances should be hidden inside one module. The access to the data inside this module is done through the set of access functions of this module. This is done to reduce the ripple effects when modifying or extending the program. Drawbacks of extensive use of modularization are the reduction of the efficiency of the whole software system and an increase in the development time. The efficiency problem can be reduced with inline access functions, which are allowed by most modern compilers.

Flexible interfaces may be a challenge in the implementation phase, but generic programming through function pointers and templates offers a solution to achieve the needed flexibility. It should be noted that the level of assumed generality should not be applied to every data structure used in the program, based on the trade-off between generality and efficiency. Certain assumptions should be made on some major data structures and a software design decision can be assumed that this data structure will not change. If such decisions are made, a detailed description of the reasons behind it should be appended to the software design documentation.

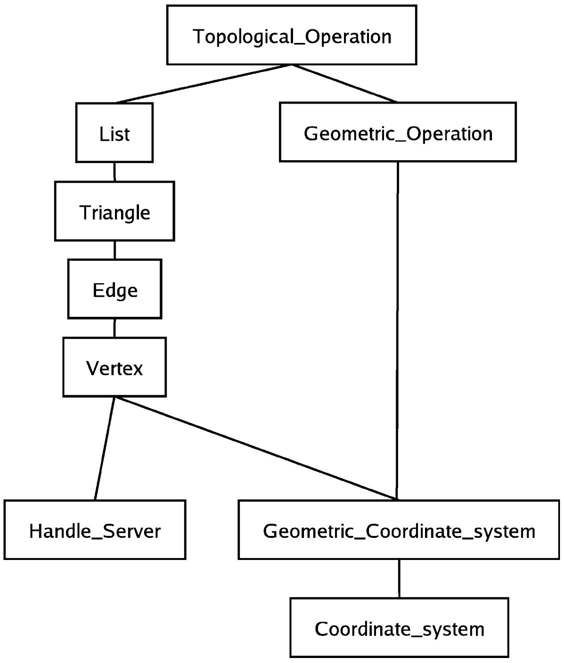

Fig. 1. A hierarchy for the designed meshing data structure.

5. A simple 2D triangular mesh data structure

The purpose of the section is not to design a complete 2D triangular mesh generator, but to demonstrate how the semi-formal specification methodology outlined can be applied to mesh generation software. The simplest modular decomposition can be found by assigning a module to each of the geometrical entities of vertex, edge and triangle. After defining these basic entities, a module for storage of these basic elements should be defined. A software design decision should be made on whether to use the same container structure for the three elements or not. After defining the basic sets of data structures, the algorithms applied on these data structures should be analyzed and divided into modules. Simple geometric operations can be contained in one module. The higher-level algorithms, which are the core of the mesh generation algorithm, should be localized in a set of independent modules because of the possibility of changing the algorithms. It should be noted that modular decomposition is not an easy job to be done in one step, instead a series of steps using stepwise refinement is applied. The previous decomposition can be represented by a uses hierarchy. We say that a module A uses a module B if correct execution of *B* may be necessary for *A* to complete its work [11]. Fig. 1 shows a uses hierarchy for the designed mesh generation software. The level of the graph shows the dependency where modules at the bottom use no other modules and considered to be at level 0. Modules at level *i* are the set of modules which use at least one module of level $i-1$ and do not use any module at level higher than $i-1$.

The goals of our 2D mesh design can be summarized as following:

(1) Having a separate and flexible representation for each mesh entity. For instance, the representation of the vertex, edge or triangle that can be easily modified or extended to accommodate different mesh generation algorithm requirements.
(2) Having a complete separation between the geometry or physical data on the mesh and the topology or

Used External Functions	:	NONE
Used External Data Types	:	NONE
External Constants	:	NONE
Type Definitions	:	Handle = int
	:	Handle_server = $(Handle)_{set}$
Exported Constants	:	MAX_SIZE: int

Exported Functions

Exported Function	**Input Type**	**Output Type**	**Exception**
HS_init	Handle_server	Handle_server	
HS_getHandle	Handle_server	Handle	Server_Is_Full
HS_addHandle	Handle_server, Handle	Handle_server	Server_Is_Full Handle_Exists
HS_delHandle	Handle_server, Handle	Handle_server	Handle_Not_Exist

State Variables: NONE

Function semantics:

Handle_server HS_init (s: Handle_server)
Output: s={}

Handle HS_getHandle (s: Handle_server)
Exception: $| s | \geq$ MAX_SIZE $\Rightarrow$ Server_Is_Full
Output: h: Handle where $h \notin s$

Handle_server HS_addHandle (s: Handle_server, h: Handle)
Exception: $| s | \geq$ MAX_SIZE $\Rightarrow$ Server_Is_Full
$h \in s \Rightarrow$ Handle_Exists
Output: $s \cup \{h\}$

Handle_server HS_delHandle (s: Handle_server, h: Handle)
Exception: $h \notin s \Rightarrow$ Handle_Not_Exist
Output: $s = s - \{h\}$

Fig. 2. Specifications of the Handle Server Module.

connectivity information of the mesh. This is done to ease the extension of the 2D mesh generator into surface meshing.

(3) The mesh generator should be able to work with different coordinate systems.
(4) A flexible data structure to store sets of vertices, edges and triangles, which can be changed based on the meshing algorithm requirements.
(5) The mesh generation can be done by different mesh generation algorithms available in the literature with a minimal amount of local changes.

The first step in our design is to define new datatypes. For example, one new datatype is introduced because of the need for each entity like the vertices, edges and triangles to have a global index or Handle. Manipulating the handle information through adding and deleting elements is not simple because of the dynamic nature of unstructured mesh generation, which allows both refinement and coarsening. Adding and removing entities during mesh generation makes the use of simple indexing infeasible. To hide the information of how to deal with indexing a Handle Server Module is defined to provide us with unique index for each of the vertices, edges and triangles. The access function of this module have a variable of type Handle server within its input parameter to provide the needed flexibility of the module to deal with three different handle servers, one for the vertices and one for edges and one for triangles. Fig. 2 shows the MIS of the Handle Server Module.

Used External Functions : NONE
Used External Data Types : NONE
External Constants : NONE
Type Definitions : CS_type = set of {2DC,2DP}
: Coord_sys = tuple of (size: int, sym: CS_type)
Exported Constants : NONE
Exported Functions :

Exported Function	**Input Type**	**Output Type**	**Exception**
CS_exists	CS_type	bool	
CS_getsize	CS_type	int	Coordsys_Not_Exist
CS_getcoordsys	CS_type	Coord_sys	Coordsys_Not_Exist
CS_getcoordtype	Coord_sys	CS_type	Coordsys_Not_Exist

State Variables: Coord_tbl : (Coord_sys) set:= {(2,2DC),(2,2DP)}
Where 2DC stands for Cartesian coordinate system (x, y) and 2DP stands for polar coordinate system (r, θ).
Function semantics:

bool CS_exists (c: CS_type)
Output: (∃ c: CS_type | (i,c) ∈ Coord_tbl)

int CS_getsize (c: CS_type)
Exception: ¬ (∃ c: CS_type | (i,c) ∈ Coord_tbl)⇒ Coordsys_Not_Exist
Output: i: int where (i,c) ∈ Coord_tbl

Coord_sys CS_getcoordsys (c: CS_type)
Exception: ¬ (∃ c: CS_type | (i,c) ∈ Coord_tbl)⇒ Coordsys_Not_Exist
Output: (i,c): Coord_sys where (i,c) ∈ Coord_tbl

CS_type CS_getcoordtype (cs: Coord_sys)
Exception: ¬ (cs ∈ Coord_tbl) ⇒ Coordsys_Not_Exist
Output: c: CS_type where (i: int, c) = cs

Fig. 3. Specifications of the Coordinate System Module.

For vertices, the handle should be combined with the geometrical data in a tuple to completely define the topology and physical information. The physical information in simple applications is limited to the geometrical data, which can be represented in many different ways. For example, the coordinate system can be Cartesian or polar. To hide the information of the coordinate system we used a Coordinate System Module as shown in Fig. 3. This module is pre-initialized with two coordinate systems, namely 2D Cartesian and 2D Polar system. This module is initialized at compilation time because of the need to define some functions to manipulate each coordinate system. The second layer of defining the geometric data is hidden in the Geometric Coordinate System Module as shown in Fig. 4. This module has the ability to manipulate information based on the specified coordinate system. An extension to this module is made by adding a set of functions for geometrical operations. Due to the large number of these geometrical operations, a separate module is defined for that purpose in Fig. 5.

Combining the handle and coordinate information for vertices is done in the Vertex Module as shown in Fig. 6. The edges can be represented explicitly as an element connecting two vertices or it can be done implicitly as the element separating two triangles. In our case, a two vertex representation is assumed because we want to keep the interface as intuitive as possible. Fig. 7 shows the MIS of the Edge Module. The triangle elements can also be represented in two ways: as three edges or by defining three vertices. It is worth mentioning that the topology or connectivity data is completely independent of whether the mesh is embedded in 2D or in 3D space as a surface mesh. Fig. 8 presents the MIS of the Triangle Module and its access functions.

Used External Functions : CS_type = set of {2DC,2DP}
: Coord_sys = tuple of (size: int, sym: CS_type)
Used External Data Types : NONE
External Constants : NONE
Type Definitions : Geometry = (real) seq
Coordinate = tuple of (cs: Coord_sys, g: Geometry)
Exported Constants : NONE
Exported Functions :

Exported Function	**Input Type**	**Output Type**	**Exception**
GE_getCoord	Coord_sys, Geometry	Coordinate	Coord_Not_Consitent
GE_setCoord	Coordinate, Geometry	Coordinate	Coord_Not_Consitent
GE_getgeom	Coordinate	Geometry	
GE_getCoordsys	Coordinate	Coord_sys	

State Variables: NONE
Function semantics:

Coordinate GE_getCoord (cs: Coord_sys, g: Geometry)
Exception: $\neg$ (cs.size = | g |) $\Rightarrow$ Coord_Not_Consistent
Output: (cs,g)

Coordinate GE_setCoord (cd: Coordinate, g: Geometry)
Exception: $\neg$ (cd.cs.size = | g |) $\Rightarrow$ Coord_Not_Consistent
Output: (cd.cs, g)

Coord_sys GE_getCoordsys (cd: Coordinate)
Output: cd.cs

Geometry GE_getgeom (cd: Coordinate)
Output: cd.g

Fig. 4. Specifications of the Geometric Coordinate System Module.

Used External Functions	:	NONE
Used External Data Types	:	Coord_sys = (size: int, sym: CS_type)
	:	Geometry = (real) seq
	:	Coordinate = tuple of (cs: coord_sys, g: Geometry)
External Constants	:	NONE
Type Definitions	:	NONE
Exported Constants	:	NONE
Exported Functions	:	

Exported Function	**Input Type**	**Output Type**	**Exception**
GO_Compute_dist	Coordinate,Coordinate	real	Coord_Not_Consistent
GO_IS_Linear	Coordinate,Coordinate, Coordinate	bool	Coord_Not_Consistent
GO_IN_Circle	Coordinate,Coordinate, Coordinate,Coordinate	bool	
GO_IN_Triangle	Coordinate,Coordinate, Coordinate,Coordinate	bool	
GO_Check_orientation	Coordinate,Coordinate, Coordinate	bool	
GO_Check_EdgeEncroach	Coordinate,Coordinate, Coordinate	bool	
GO_Check_TriEncroach	Coordinate,Coordinate, Coordinate,Coordinate	bool	
GO_Get_EdgeMid	Coordinate,Coordinate	Coordinate	
GO_GetCircumcenter	Coordinate,Coordinate, Coordinate	Coordinate	

State Variables: NONE
Selected function semantics:

```
real GO_Compute_dist (c1: Coordinate, c2: Coordinate)
        Exception:   ¬ (GE_getCoordsys(c1)= GE_getCoordsys (c2)) ⇒
                     Coord_Not_Consistent
        Output:      If CS_getcoordtype(GE_getCoordsys(c1))=2DC
                     GE_Compte_dist2DC(c1.g,c2.g)
                     If CS_getcoordtype(GE_getCoordsys(c1))=2DP
                     GE_Compte_dist2DP(c1.g,c2.g)
```

Internal function semantics:

```
real GO_Compute_dist2DC (g1: Geometry, g2: Geometry)
        Output:      Sqrt((g1[0]-g2[0])ˆ2+(g1[1]-g2[1])ˆ2)

real GO_Compute_dist2DP (g1: Geometry, g2: Geometry)
        Output:      Sqrt(g1[0]ˆ2+ g2[0]ˆ2-2*g1[0]* g2[0]* cos(g2[1]–g1[1]))
```

Fig. 5. Excerpts from specifications of the Geometric Operation Module.

The next step is to define the container specifications of each entity. A generic container specification is shown in Fig. 9. A type variable, which may be a vertex, edge or triangle is used in this specification. A specialization of this list or container is done to have the VertexList and the EdgeList and the TriangleList. Finally, a set of some topological operation commonly used by unstructured mesh generation algorithms are bundled in the Topological Operation Module shown in Fig. 10.

6. Specifications of mesh generation algorithms

Mesh generation algorithms can be specified using the developed infrastructure. Mesh generators usually needs two types of relation between mesh entities. These relations can be divided into incidence and adjacency relations. Betri [2] formalized the definition of the *incidence* relation into the relation of a subset. If a mesh entity *f* is inside of another entity *c* then *f* and *c*

Used External Functions	:	NONE
Used External Data Types	:	Coordinate = tuple of (cs: coord_sys, g: Geometry)
	:	Handle = int
External Constants	:	NONE
Type Definitions	:	Vertex = tuple of (hd: Handle, cd: Coordinate)
Exported Constants	:	NONE

Exported Functions

Exported Function	**Input Type**	**Output Type**	**Exception**
V_createVertex	Handle, Coordinate	Vertex	
V_getHandle	Vertex	Handle	
V_getCoord	Vertex	Coordinate	
V_Compare	Vertex, Vertex	bool	
V_setVertCoord	Vertex, Coordinate	Vertex	

State Variables: NONE

Function semantics:

Vertex V_createVertex (h: Handle, cd: coordinate)
Output: (h,cd)

Handle V_getHandle (v: Vertex)
Output: v.hd

Coordinate V_getCoord (v: Vertex)
Output: v.cd

bool V_Compare (v1: Vertex, v2:Vertex)
Output: v1.hd = v2.hd

Vertex V_setVertCoord (v: Vertex, cd: coordinate)
Output: (v.hd,cd)

Fig. 6. Specifications of the Vertex Module.

are incident. For example, there is an incidence relation between the start vertex of an edge and the edge itself. It is clear that elements of the same topological dimension are never incident, but they may have another type of relation called *adjacency* relation. For example, we can define that two edges are *adjacent* if they share the same vertices. The incident relations are specified in the mesh infrastructure sections, where a downward incidence relation from elements with higher topological dimensions are connected to elements with only 1D less in the topological sense. Thus, triangles are defined in terms of edge and edges are defined in terms of vertices. On the other hand, the adjacency relation was identified as being algorithm dependant. For example, Oct-tree based mesh generators rely on parent/child adjacency relations between entities of the same topological dimension, while in Delaunay triangulation each triangle needs to know the neighboring triangles through the neighbor adjacency relation. Due to this dependency of the adjacency relations on the mesh generation algorithm, these relations are not defined in the mesh infrastructure.

As an example of using the suggested semi-formal documentation and specification style, a key operation of a Delaunay mesh generation algorithm is specified. Delaunay triangulation is one of the most common algorithms for triangular mesh generation. These algorithms are usually done incrementally, where an initial large triangle that geometrically bounds all the domain is defined. Following this, vertices along the boundaries are inserted incrementally. Once all the boundary vertices are inserted, boundary edges are recovered. The recovery is

Used External Functions	:	NONE
Used External Data Types	:	Vertex = tuple of (hd: Handle, cd: Coordinate)
	:	Handle = int
External Constants :	:	NONE
Type Definitions:	:	Edge : tuple of (hd: Handle, v0: Vertex, v1: Vertex)
Exported Constants:		NONE

Exported Functions:

Exported Function	**Input Type**	**Output Type**	**Exception**
E_createEdge	Handle, Vertex, Vertex	Edge	Edge_Not_Valid
E_getHandle	Edge	Handle	
E_Compare	Edge, Edge	bool	
E_getStart	Edge	Vertex	
E_getEnd	Edge	Vertex	
E_setEdge	Edge, Vertex, Vertex	Edge	Edge_Not_Valid

State Variables: NONE

Function semantics:

```
Edge E_createEdge (h: Handle, v0: Vertex, v1: Vertex)
          Exception:    V_Compare(v0,v1) ⇒ Edge_Not_Valid
          Output:       (h,v0,v1)

Handle E_getHandle (e: Edge)
          Output:       e.hd

bool E_Compare (e1: Edge, e2: Edge)
          Output:       e1.hd = e2.hd

Vertex E_getStart (e: Edge)
          Output:       e.v0

Vertex E_getEnd (e: Edge)
          Output:       e.v1

Edge E_setEdge (e: Edge, v0: Vertex, v1: Vertex)
          Exception:    V_Compare(v0,v1)⇒ Edge_Not_Valid
          Output:       (e.hd,v0,v1)
```

Fig. 7. Specifications of the Edge Module.

also done by inserting vertices along the missing boundaries. Finally a mesh improvement by refinement is done for all the triangles that do not meet a certain quality measure. A new vertex is inserted at the circumcenter of each triangle that fails the geometrical quality predicate. A complete description of the Delaunay refinement algorithms can be found in Ref. [18]. It is clear from the previous description that vertex insertion is the core step of this algorithm. This insertion should maintain the validity of the Delaunay empty circumcenter property of every triangle in the mesh. Fig. 11 introduces the specifications of neighbor adjacency relation of the mesh edges. This relation is needed for the Delaunay refinement algorithms to identify adjacent triangles. Additional adjacency relations can be defined in the implementation process, but if any redundancy in the stored information is introduced, validity checks should also be added to avoid any inconsistency. A pictorial representation of the Bower/Watson point insertion algorithm [4,19] is shown in Fig. 12. In this

Used External Functions : NONE
Used External Data Types : Handle = int
: Edge = tuple of (hd: Handle, v0: Vertex, v1: Vertex)
External Constants : : NONE
Type Definitions: : Triangle = tuple of (hd: Handle, e0: Edge, e1: Edge, e2: Edge)
Exported Constants: NONE
Exported Functions:

Exported Function	**Input Type**	**Output Type**	**Exception**
T_createTri	Handle, Edge, Edge, Edge	Triangle	Triangle_Not_Valid
T_getHandle	Triangle	Handle	
T_Compare	Triangle, Triangle	bool	
T_getE0	Triangle	Edge	
T_getE1	Triangle	Edge	
T_getE2	Triangle	Edge	
T_isEdge	Triangle, Edge	bool	
T_setTriangle	Triangle, Edge, Edge, Edge	Triangle	Triangle_Not_Valid

State Variables: NONE
Function semantics:

```
Triangle T_createTri (h: Handle, e0: Edge, e1: Edge, e2:Edge)
        Exception:  E_Compare(e0,e1) ∨ E_Compare(e1,e2) ∨ E_Compare(e2,e0)
                    ⇒ Triangle_Not_Valid
        Output:     (h,e0,e1,e2)

Handle T_getHandle (t: Triangle)
        Output:     t.hd

bool T_Compare (t1: Triangle, t2: Triangle)
        Output:     t1.hd = t2.hd

Edge T_getE0 (t: Triangle)
        Output:     t.e0

Edge T_getE1 (t: Triangle)
        Output:     t.e1

Edge T_getE2 (t: Triangle)
        Output:     t.e2

bool T_isEdge (t: Triangle, e: Edge)
        Output:     E_Compare(t.e0,e) ∨ E_Compare(t.e1,e) ∨ E_Compare(t.e2,e)

Triangle T_setTriangle (t: Triangle, e0: Edge, e1: Edge, e2: Edge)
        Exception:  E_Compare(e0,e1) ∨ E_Compare(e1,e2) ∨ E_Compare(e2,e0)
                    ⇒ Triangle_Not_Valid
        Output:     (t.hd,e0,e1,e2)
```

Fig. 8. Specifications of the Triangle Module.

algorithm, whenever a new vertex is inserted all the triangles where the new vertex falls within its circumcircle (encroached) are deleted. The new cavity is then triangulated by connecting the new vertex to the vertices on the boundary of the resulting cavity. Fig. 13 presents the specifications of point insertion as a part of the Bower/Watson algorithm for Delaunay triangulations.

Used External Functions : NONE
Used External Data Types : Handle = int
: Entity = a
where a is a type variable which
is one of three type Vertex, Edge or Triangle.
: List = (Entity) set
External Constants : NONE
Type Definitions: : NONE
Exported Constants: : MAX_SIZE: int
Assumption: : List is initialized before usage.
Exported Functions:

Exported Function	**Input Type**	**Output Type**	**Exception**
L_size	List	int	
L_addObj	Entity, List	List	List_Is_Full, Obj_Exists
L_delObj	Entity, List	List	Obj_Not_Exist
L_clear	List	List	

State Variables: NONE
Function semantics:

int L_size (l: list)
Output: $| l |$

List L_addObj (e: Entity, l: List)
Exception: $| l | \geq$ MAX_SIZE $\Rightarrow$ List_Is_Full
$\exists$ e1: Entity $\in$ l where e1.hd = e.hd $\Rightarrow$ Obj_Exists
Output: $l \cup \{e\}$
List L_delObj (e: Entity , l: List)
Exception: $e \notin l \Rightarrow$ Obj_Not_Exist
Output: $l - \{e\}$

List L_clear (l: List)
Output: $l = \{\}$

Fig. 9. Specifications of the List Module.

7. Extendability and scalability

The extendability of the introduced mesh generation system is granted by our modularization. For example, Oct-tree based meshing algorithms do not share many operations with Delaunay based algorithms, but our meshing system can be extended to Oct-tree algorithms in a straight forward way. Oct-tree mesh generation requires a tree structure to define the adjacency between the mesh entities. This tree structure will be specified as a variation of the adjacency relation module. The mesh generation algorithm can be considered as a variation of the Delaunay insertion algorithm where vertices are inserted incrementally with different criterion to maintain the tree balancing. Once a node is inserted inside a triangle that includes another vertex, that triangle should be divided into a pre-specified number of children followed by a tree-balancing step. Boundary recovery will also depend on inserting new vertices. This demonstrates that to adopt a completely different mesh generation algorithm only two modules need to be changed. These two modules are the adjacency relation module and the mesh generation module.

The scalability of this meshing system is assumed to be similar to the development of matrix analysis libraries BLAS [9] and LAPACK [1]. The BLAS library provides the basic vector and matrix operation on different data

Used External Functions	:	NONE
Used External Data Types	:	Handle_server = (Handle) set
	:	Vertex = (hd: Handle, cd: Coordinate)
	:	Edge = (hd: Handle, v0: Vertex, v1: Vertex)
	:	Triangle = (hd: Handle, e0: Edge, e1: Edge, e2: Edge)
	:	list = (Entity) set
External Constants	:	NONE
Type Definitions	:	NONE
Exported Constants	:	NONE
Selected Exported Functions	:	

Exported Function	**Input Type**	**Output Type**	**Exception**
TO_splitEdge	Edge		
TO_InsVertOnEdge	Vertex, Edge		Vert_Not_OnEdge

State Variables:	VertexList: list :=(Vertex) set
	EdgeList: list :=(Edge) set
	TriangleList: list :=(Triangle) set
	Vhandle_server: Handle_server
	Ehandle_server: Handle_server
	Thandle_server: Handle_server

Selected Function semantics:

```
TO_splitEdge (e: Edge)
            Transition:  v1_temp = E_getStart(e: Edge)
                         v2_temp = E_getEnd(e: Edge)
                         cd1_temp = V_getCoord(v1_temp)
                         cd2_temp = V_getCoord(v2_temp)
                         cd3_temp = GO_GetEdgeMid(cd1_temp, cd2_temp)

                         v3hd_temp = HS_getHandle(Vhandle_server)
                         Vhandle_server =HS_addHandle(Vhandle_server, v3hd_temp)
                         v3_temp = V_createVertex(v3hd_temp,cd3_temp)
                         VertexList = L_addObj(v3_temp, VertexList)

                         e1hd_temp = E_getHandle(e)
                         e1_temp = E_createEdge(e1hd_temp, v1_temp, v3_temp)
                         e2hd_temp = HS_getHandle(Ehandle_server)
                         Ehandle_server=HS_addHandle(Ehandle_server, e2hd_temp)
                         e2_temp = E_createEdge(e2hd_temp, v3_temp, v2_temp)

                         EdgeList = L_delObj(e,EdgeList)
                         EdgeList = L_addObj(e1_temp,EdgeList)
                         EdgeList = L_addObj(e2_temp,EdgeList)
```

Fig. 10. Specifications of the Topological Operation Module.

types and LAPACK provides high level routines for different problems like the solution of linear equations, singular value decomposition and many other problems. The newly introduced mesh generation infrastructure is similar in concept to the BLAS library and has been divided into a storage scheme, topological and geometrical operation at the lowest level. On top of this different mesh generation algorithms can be developed. These algorithms can interface cleanly with different storage and data access schemes. Finally, the high level mesh based applications, such as the finite element method can utilize the entire infrastructure.

Used External Functions	:	NONE
Used External Data Types	:	Handle = int
	:	Edge = (hd: Handle, v0: Vertex, v1: Vertex)
		EAdj_List = (hd0: Handle, hd1: Handle) set
External Constants	:	NONE
Type Definitions:	:	NONE
Exported Constants:	:	NONE
Assumption:	:	Adj_List is initialized before usage.

Exported Functions:

Exported Function	**Input Type**	**Output Type**	**Exception**
EA_addpair	Edge, Edge, EAdj_List	EAdj_List	Edge_AlreadyIn_Pair
EA_delpair	Edge, EAdj_List	EAdj_List	Edge_Not_In_Pair
EA_IsAdj	Edge, EAdj_List	Bool	
EA_getAdj	Edge, EAdj_List	Edge	Edge_Not_In_Pair
EA_clear	List	List	

State Variables: NONE

Function semantics:

```
Bool        EA_IsAdj (e: Edge, a: EAdj_List)
            Output:     ( (e0.hd, hd:Handle) ∧ (hd:Handle, e0.hd) ) ∈ a )
EAaj_List   EA_addPair (e0: Edge, e1: Edge, a: EAdj_List)
            Exception:  ( EAdj_IsAdj(e0,a) ∨ EAdj_IsAdj(e1,a))
                        ⇒ Edge_AlreadyIn_Pair
            Output:     a ∪ (e0.hd, e1.hd) ∪ (e1.hd, e0.hd)
EAdj_List   EA_delEdge (e: Edge, a: EAdj_List)
            Exception:  (e.hd, hd:Handle) ∉ a ∨ (hd:Handle, e.hd) ∉ a ⇒ Edge_Not_In_Pair
            Output:     a – {(e.hd, hd:Handle),(hd:Handle, e.hd)}
```

Fig. 11. Specifications of the edge Adjacency List Module.

8. Conclusions

Using the specified 2D mesh generation infrastructure reliable mesh generation software can be developed in a simple way for any mesh generation algorithm. The clear high level description of the basic entities of the mesh and the complete separation between the topological and geometrical information makes it easy to extend and modify this tool. The high level of abstraction of the containers as sets leaves the selection of an efficient representation for storage of mesh entities until a decision about the meshing algorithm is taken. This can be done in the next step of specification refinement, or it can be left for the implementation phase. The specification presented can significantly help in avoiding any ambiguity during the design process of mesh generation software. Writing the design specifications in a formal way, which is intended for humans and eventually for machine

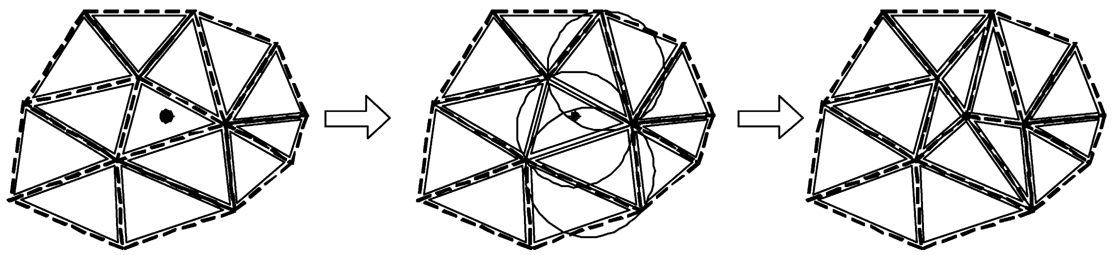

Fig. 12. Bower/Watson point insertion algorithm.

Used External Functions	:	NONE
Used External Data Types	:	Handle_server = (Handle) set
	:	Vertex = (hd: Handle, cd: Coordinate)
	:	Edge = (hd: Handle, v0: Vertex, v1: Vertex)
	:	Triangle = (hd: Handle, e0: Edge, e1: Edge, e2: Edge)
	:	list = (Entity) set
	:	EAdj_list = (hd0: Handle, hd1: Handle) set
External Constants	:	NONE
Type Definitions	:	NONE
Exported Constants	:	NONE
Selected Exported Functions	:	

Exported Function	**Input Type**	**Output Type**	**Exception**
D_InsVert	Vertex		

State Variables: VertexList: list :=(Vertex) set; EdgeList: list :=(Edge) set
TriangleList: list :=(Triangle) set
Vhandle_server: Handle_server; Ehandle_server: Handle_server
Thandle_server: Handle_server

Selected Function semantics:

```
D_InsVert (v: Vertex)
        Transition:  Del_TriList_temp = { t: Triangle | GO_Check_TriEncroach(
                     V_getCoord(E_getStart(t.e0)),V_getCoord(E_getStart(t.e1)),
                     V_getCoord(E_getStart(t.e2)), V_getCoord(v) ) }

                     Del_EdgeList_temp = { e: Edge | T_isEdge(t,e) ∧ t ∈ Del_TriList_temp }

                     Replaced_EdgeList_temp = { e: Edge | e ∈ Del_EdgeList_temp ∧
                     EA_getAdj(e) ∉ Del_EdgeList_temp }

                     for all e ∈ Replaced_EdgeList_temp DO D_CreateTri(v, e: Edge)
                     EdgeList = EdgeList – { Del_EdgeList_temp – Replaced_EdgeList_temp }
                     TriangleList = TriangleList – Del_TriList_temp
```

Internal function semantics:

```
D_CreateTri (v: Vertex, e: Edge)
        Transition:  e1hd_temp = E_getHandle(e)
                     Ehandle_server=HS_addHandle(Ehandle_server, e2hd_temp)
                     e1_temp = E_createEdge(e1hd_temp, E_getStart(e), v)

                     e2hd_temp = HS_getHandle(Ehandle_server)
                     Ehandle_server=HS_addHandle(Ehandle_server, e2hd_temp)
                     e2_temp = E_createEdge(e2hd_temp, E_getEnd(e) , v)

                     EdgeList = L_addObj(e1_temp,EdgeList)
                     EdgeList = L_addObj(e2_temp,EdgeList)

                     thd_temp = HS_getHandle(Thandle_server)
                     Thandle_server =HS_addHandle(Thandle_server, thd_temp)
                     t_temp = T_createTri(thd_temp, e, e1_temp ,e2_temp)
                     TriangleList = L_addObj(t_temp,TriangleList)
```

Fig. 13. Excerpts from Delaunay vertex Insertion Module.

verification is considered by the authors as a very reliable method. Identifying exception cases early and defining the proper action to be taken protects the software design from major changes at the testing stages. The complete specification with all exception cases defined will significantly help in driving test cases to check the correctness of the final product as well as for testing each module separately.

References

[1] Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Croz JD, Greenbaum A, Hammarling S, Mckenney A, Sorensen D. LAPACK users' guide, 3rd ed. Philadelphia, PA: SIAM; 1999.

[2] Berti G. Generic software components for scientific computing. PhD Thesis. BTU Cottbus, Germany: Faculty of Mathematics, Computer Science, and Natural Science; 2000.

[3] Berti G. GrAL—the grid algorithms library. Lect Notes Comput Sci 2002;2331:745–54.

[4] Bowyer A. Computing dirichlet tessellations. Comput J 1981;24: 162–6.

[5] Chen C-H. A software engineering approach to developing a mesh generator. Master's Thesis. Hamilton, Ont.: McMaster University; 2003.

[6] Glinz M. Problems and deficiencies of UML as a requirements specification language Proceedings of the 10th International Workshop on Software Specification and Design.: IEEE Computer Society; 2000 p. 11.

[7] Hoffman D, Strooper P. Software design, automated testing, and maintenance. A practical approach.: International Thomson Publishing; 1995.

[8] Huth MR, Ryan MD. Logic in computer science: modelling and reasoning about systems. Cambridge: Cambridge University; 2000.

[9] Lawson CL, Hanson RJ, Kincaid DR, Krogh FT. Basic linear algebra subprograms for fortran usage. ACM Trans Math Software, 5 1979; 3(3):308–23.

[10] Object Management Group. OMG unified modeling language specification 2003. Version 1.5.

[11] Parnas DL. On a 'buzzword': hierarchical structure Proceedings of the IFIP 74.: North Holland Publishing Company; 1974 p. 336–339.

[12] Parnas DL, Clement PC, Weiss DM. The modular structure of complex systems International Conference on Software Engineering 1984 p. 408–419.

[13] Parnas DL, Weiss DM, Hoffman D. Software fundamentals: collected papers. In: Parnas DL, editor.. Reading, MA: Addison-Wesley; 2001.

[14] Peled DA. Software reliability methods. Berlin: Springer; 2001.

[15] Piff M. Discrete mathematics: an introduction for software engineers. Cambridge: Cambridge University; 1991.

[16] Remacle J-F, Shephard MS. An algorithm oriented mesh database. Int J Numer Meth Eng 58-2 2003;349–74.

[17] Owre S, Rajan S, Rushby JM, Shankar N, Srivas MK. PVS: combining specification, proof checking, and model checking. In: Alur R, Henzinger TA, editors. Proceedings of the Eighth International Conference on Computer Aided Verification CAV (New Brunswick, NJ, USA/1996), vol. 1102. Berlin: Springer; 1996, p. 411–4.

[18] Shewchuk JR. Delaunay refinement mesh generation. PhD Thesis. Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, May; 1997. Available as Technical Report CMU-CS-97-137.

[19] Watson DF. Computing the *n*-dimensional Delaunay tessellation with application to Voronoi polytopes. Comput J 1981;24:167–71.