IMPROVING THE QUALITY OF A PARALLEL MESH GENERATION TOOLBOX

A DOCUMENT DRIVEN METHODOLOGY FOR IMPROVING THE QUALITY OF A PARALLEL MESH GENERATION TOOLBOX

By

WEN YU, B.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2007) McMaster University

COMPUTING AND SOFTWARE Hamilton, Ontario

TITLE: A Document Driven Methodology for Improving The Quality of a Parallel Mesh Generation Toolbox

AUTHOR: Wen Yu, B.Sc. (McMaster University)

SUPERVISOR: Dr. Spencer Smith

NUMBER OF PAGERS: xxii, 216

Abstract

Scientific computing software has had considerable success in producing efficient and correct numerical results. However other software qualities, such as usability, maintainability, testability, flexibility, and reusability, are often neglected. Presented in this work is our proposed solution to improve the quality of scientific computing software by using a document driven software engineering methodology. A parallel mesh generation toolbox (PMGT) is developed to illustrate our approach.

This thesis proposes to improve quality via a methodology that consists of a sequence of design steps and documents, including the following: a Software Requirements Specification (SRS), a Module Guide (MG), a Module Interface Specification (MIS), and a Summary of Validation Testing Report (SVTR). Where applicable, mathematical notation is used in these documents to make them as formal as possible. This formality improves the documents by making them less ambiguous and more validatable; therefore, the correctness and testability of the software are improved. The proposed methodology also requires that the traceability between the documents listed above, and the traceability between these documents and code be explicitly specified. This allows for verification of completeness and consistency and facilitates systematic change management.

Quality is also promoted during the implementation stage. For instance, a new modification is proposed to Rivara's longest side bisection algorithm. The modified algorithm improves the quality of usability, without

sacrificing reliability. A new coarsening algorithm inspired by Oliver-Gooch is also proposed. Instead of decimating vertices by collapsing the edges, the new algorithm uses edge collapse to decimate the cells.

The proposed methodology promotes testing as an important way to improve software quality. However, due to the lack of an expected answer, testing the correctness of PMGT is difficult. To overcome this challenge, the method promoted in our work is automated testing to verify the known properties of a correct solution, such as checking for conformality and for boundary closure.

Acknowledgements

First of all, I would like to express my sincere thanks and deep appreciation to Dr. Spencer Smith, my supervisor, for his constant support and encouragement. He shared so many great ideas with me and carefully corrected my mistakes and typos. This thesis would not be what it is without him. He is one of the nicest professors I have ever met.

I am grateful to Dr. Qiao and Dr. Khedri for reviewing of this thesis and giving me valuable feedbacks and suggestions.

Of course, I am thankful to my parents for their support and endless love. I wish their happiness and good health. I hope that they enjoy their retirement life in China.

Thanks to (alphabetically) Fang Cao, Huarong Chen, Ahmed H. ElSheikh, Dai Tri Man Le, John McCutchan, Jin Tang, Shu Wang, Qian Yang, Munira Yusufu, Yun Zhai, Haijun Zou, and many others in the Computing and Software department, for their friendship.

Last but not least, I would like to give my special thanks to my husband, Kelvin, and my son, Mike, for their care and support. They have been such a blessing in my life.

Hamilton, Ontario, Canada Wen Yu

January, 2007

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many physical problems of importance to scientists and engineers are modeled as a set of Partial Differential Equations (PDEs). In most practical cases, it is necessary to solve the PDEs numerically. Numerical methods to solve PDEs frequently require that the domain of interest be divided into a mesh, which is a set of small, simple elements (shapes) that cover the computational domain. In some applications, a single mesh is generated and used many times; in this case, the processing time spent on mesh construction is not critical and a relatively slow, sequential algorithm suffices (Ruppert, 1993). However, some applications need adaptive meshing, which requires that the meshes be generated once and then modified many times. For instance, adaptive meshing, which involves many mesh changes, is used for reliable Finite Element Analysis (FEA) using a posterrori error estimation (Zienkiewicz et al., 2005). The increased mesh interaction for adaptive meshing means an increased need for speed in managing the mesh data. This suggests employing parallel processing

techniques. Although generating a mesh using multiple processors is complicated, it can offer considerable speed-up over sequential processing. In addition, some FEA applications are implemented on multiple processors. If the adaptive mesh can be generated in multiple processors as well, the mesh data can remain on the local processors. Using local processors in this way has the potential to significantly reduces overall computation time.

While considerable effort has been spent on research on mesh generation algorithms to improve efficiency and correctness, other qualities of mesh generation software, such as usability, maintainability, testability, flexibility, portability, and reusability, can still be improved. Software engineering methodologies have been adopted successfully across a broad spectrum of industry applications to achieve high quality software. However, software engineering methodologies are rarely applied to developing scientific computing software, including mesh generation software. This thesis addresses this past neglect by motivating, justifying and illustrating how the quality of a parallel mesh generation toolbox (PMGT) can be improved by using software engineering methodologies.

This chapter provides introductory information about the thesis. Software quality is defined in Section 1.1. The characteristics of scientific computing software that contribute to the challenge of developing quality software are summarized in Section 1.2. Current mesh generation tools are investigated and summarized in Section 1.3. The basic knowledge of software engineering methodologies is provided in Section 1.4. Finally, the proposed methodology for developing PMGT is introduced in Section 1.5.

1.1 What is Quality Software?

Quality of software is often defined as "meeting requirements" (Lewis and Veerapollai, 2004; Copeland, 2003; CSTE, 2006). With this meaning, quality is a binary state; that is, it is a quality product or not. However, this definition has its limitations. A limitation of this definition can be seen by considering two different software programs, s_1 and s_2. Program s_1 can perform a task in one hour, while s_2 can perform the same task in two hours. All other features of two program are assumed to be the same. Intuitively, s_1 would be said to have higher quality than s_2, even if *Efficiency* is not a requirement of the software. However, using the definition of "meeting requirements," s_1 and s_2 would be of the same quality. This is not the consequence that one would expect. In addition, a binary value should not be used to define the quality of software. The same example software s_1 and s_2 can be used to illustrate the limitation of a binary definition. Suppose the requirement of *Efficiency* is specified as "execution time of the software should be less than three hours." Then, by the binary definition, the quality of the two example programs is the same, but this contradicts our intuition.

Pressman (1999) and Ghezzi et al. (2003) give different points of view from the "meeting requirements" definition given above about the quality of software. Both definitions include two categories of quality, which are product qualities and process qualities. The product qualities are measured by how well the software conforms to both explicit requirements, which are "requirements" from users, and implicit requirements, such as the desire for good maintainability. The classical quality factors proposed by McCall et al. (1997)

Factors	**Definition**
Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives.
Reliability	Extent to which a program can be expected to perform its intended function with the required precision.
Efficiency	The amount of computing resources and code required by a program to perform a function.
Integrity	Extent to which access to software or data by unauthorized persons can be controlled.
Usability	Effort required for learning, operating, preparing input, and interpreting the output of a program.
Maintainability	Effort required for locating and fixing an error in an operational program.
Testability	Effort required in testing a program to ensure that it performs its intended function and how well the program performs its function.
Flexibility	Effort required in modifying an operational program.
Portability	Effort required to transfer software from one configuration to another.
Reusability	Extent to which a program can be used in other applications – related to the packaging and scope of the functions that programs perform.
Interoperability	Effort required to couple one system with another.

Table 1.1: Software Quality Factors. Table modified from McCall et al. (1997)

that are used to measure the qualities of software are listed in Table 1.1. Although the table lists the qualities of a program, these qualities will also apply to other software products, such as a source code library. In the case of a source code library, the definitions would have to be slightly modified. For instance, usability would now refer to the effort required for a programmer to use the library.

The definition of quality for software products adopted in this thesis is as follows:

> The quality of a software product is the degree to which the software conforms to the software quality factors.

Since the common "requirements" of software include one or more software quality factors, this definition is, in fact, an extension of the definition of "meet requirements." The "requirements" may not include all the factors. However, the factors still provide a assessment of the quality. The "requirements" provide the stakeholders' judgment on which of the qualities are most important and how to measure and evaluate whether the important qualities have been met. Achieving product quality is the ultimate goal. However, a process must be followed to improve the chance of achieving product quality. Studying the quality of the process is outside the scope of this thesis.

1.2 Challenges in Scientific Computing Software

The quality factors listed in Table 1.1 are general. These factors do not have equal importance between different types of software. For example, *Efficiency* (QF3) in time is critical for a real-time application. However, it is usually not as important for a word processing application. The important factors for scientific computing software are proposed to be the following:

- QF1: Correctness
- QF2: Reliability
- QF3: Efficiency

- QF4: Usability
- QF5: Maintainability
- QF6: Testability
- QF7: Flexibility
- QF8: Portability
- QF9: Reusability

Scientific computing software, as a special class of software, has its own characteristics. Some of the characteristics that make achieving the above qualities a challenge for scientific computing software are summarized below.

1. **Unknown Solution Challenge**

   The answers for most scientific computing problems are unknown. Most scientific computing software is built to solve problems that are difficult or impossible to solve without the software. Hence, the software is the only possible way that the solution to the problem can be achieved. Judging the *Correctness* (QF1) of scientific computing software is more difficult than for other classes of software due to the lack of expected answers.

2. **Real Number Representation Challenge**

   Most real numbers cannot be represented exactly on a computer. Floating point numbers are used to approximate real numbers. However, this approximation can cause problems. A well-known example is the

American Patriot Missile battery in Dharan, Saudi Arabia that failed to track and intercept an incoming Iraqi Scud missile during the Gulf War (Cirincione, 1992). The challenge of approximating real numbers makes it difficult to achieve *Correctness* (QF1) and *Reliability* (QF2). Using more storage for floating point numbers can help to some extent because it improves the precision of the computer representation of the real numbers. However, this decision has a tradeoff as it can potentially lower the *Efficiency* (QF3) of the software.

3. **Nonfunctional Requirement Challenge**

   As for other classes of software, for scientific computing software, nonfunctional requirements are as important as functional requirements, and nonfunctional requirements are difficult to properly specify and measure. For example, it is difficult to specify the usability requirement of software. This challenge will be explained in greater detail in Chapter 2.

4. **Parallel Computation Challenge**

   Scientific computing problems often deal with large amounts of calculation. Some scientific computing software takes advantage of parallel computation to improve the *Efficiency* of the software. However, using multi-processor usually lowers the *Usability* (QF4), *Maintainability* (QF5), and *Portability* (QF8), since communication between processors must be considered. The *Reliability* (QF2) can potentially be reduced because of the errors introduced during the communication between processors.

1.3 Mesh Generation Tools

Meshing is the process of decomposing a spatial domain into smaller and simpler elements. Common shapes of elements are triangles and quadrilaterals for a two dimensional domain, and tetrahedra or hexahedra for a three dimensional domain. Since the shape of the domain of the mesh may be irregular, unstructured meshes, which can discretize the domain more naturally than structured meshes, are of particular interest. An example mesh of Lake Superior is shown in Figure 1.1.

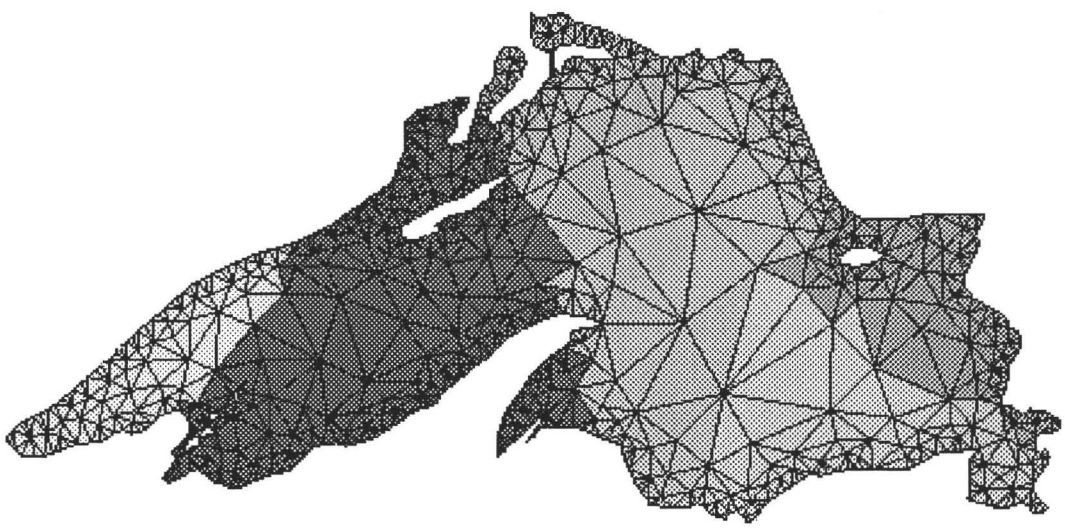

Figure 1.1: A Mesh of Lake Superior. Image from Shewchuk (Last Access: January, 2006)

Owen (1998) surveyed 94 mesh generation software packages. Most of the software on his list was developed by the people who intend to use it. The advantage of this is that they are experts in the application area; hence, they understand the requirements of the software well. However, they usually lack knowledge of software methodologies, which can cause problems in the

current approach to developing mesh software. First, mesh generation software is often developed by modifying an existing program. This approach demonstrates the importance of the requirement of reuse for mesh generation software. However, this "copy and paste" method for code reusing results in the growth of the software in unexpected ways. It is often the situation that the existing code has more functionality than one expects or desires. By adding more functionality, the code becomes bigger and bigger. This makes achieving the quality of *Efficiency* (QF3) difficult. A second problem with the current approach to developing mesh generation software is that it results in many similar mesh generation software packages. For instance, of the 94 software packages surveyed by Owen (1998), 61 of them generate triangle meshes, and 43 of them use the Delaunay Algorithm. These numbers illustrate that although the requirement of reuse exists in mesh generation software development, it is not fulfilled very well. The fact is that *Reusability* (QF9) of current mesh generation software is rarely achieved. The third problem with the current approach is that the documentation of many mesh generators is incomplete, ambiguous, or even non-existent (Cao, 2006). Cao (2006) observed that among 120 papers available on Owen (Last Access: January, 2006) from 2002 to 2004, only 3 papers talk about the design of mesh generators. Without proper documentation, software is not only difficult to understand and maintain, it is also hard to validate and extend. As a consequence, the resulting software has poor quality in terms of *Usability* (QF4), *Maintainability* (QF5), *Testability* (QF6), and *Reusability* (QF9).

1.4 Software Engineering Methodologies

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software (IEEE, 1990). Using software engineering methodologies that relate to software development can improve the quality of software. The debate by software developers has switched from whether software engineering methodologies should be used to which methodologies are best for software development.

All software development can be characterized as a problem solving loop in which four distinct stage are encountered: status quo, problem definition, technical development, and solution integration (Pressman, 1999). Pressman (1999) gives explanations of each stage as follows: *i*) Status quo "represents the current state of affairs" (Raccoon, 1995); *ii*) problem definition identifies the specific problem to be solved; *iii*) technical development solves the problem through the application of some technology; and, *iv*) solution integration delivers the results to those who requested the solution in the first place. The development strategy that software engineers use is often referred to as a process model. Among many models proposed by software engineering researchers, the *waterfall model, prototype model*, and *evolutionary model* attract much of the attention.

This section gives an introduction to the above models. Determining the development model is an important decision that will effect all other subsequent decisions. Other aspects of software engineering methodologies will be introduced in the rest of the thesis.

1.4.1 The Waterfall Model

The waterfall model was originally proposed by Royce (1970). The major stages of this model are requirements, design, implementation, verification, and maintenance, as illustrated in Figure 1.2. This model treats the process of developing software as a sequence of stages; therefore, the waterfall model is sometimes called the *linear sequential model*. The criteria of finishing a stage is the completion of the documentation for this stage. The waterfall model was the first systematic approach to developing software. It is a widely used model and is still the reference model for most software engineering textbooks and standard industry practices (Pressman, 1999; Ghezzi et al., 2003). However, this model has its difficulties. Lack of feedback is its most notable disadvantage.

1.4.2 The Prototype Model

The prototyping model emphasizes communication between customers and software developers. The prototype developers build services as a mechanism for identifying software requirements. There is no requirement for the quality of the prototype, and the prototype should be discarded if it does not meet the software quality criteria. However, in many cases, the poor quality prototype is built and carried forward to become real software.

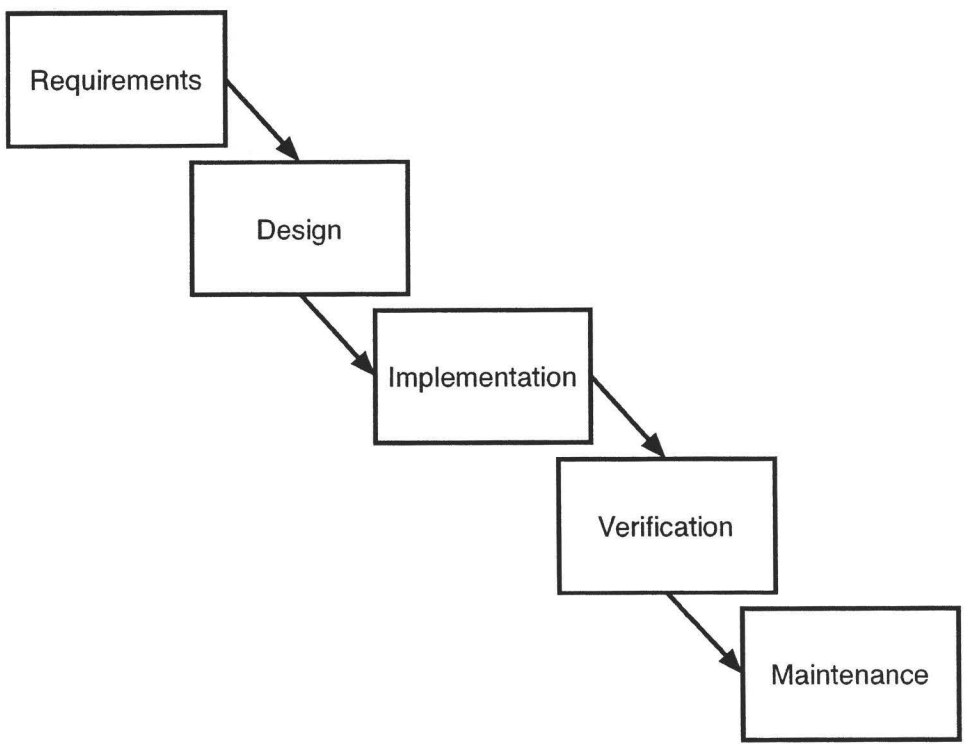

Figure 1.2: Waterfall Model

1.4.3 The Evolutionary Development Model

The evolutionary model is based on an observation that requirements often change as development proceeds. Extending the waterfall model, the evolutionary model incorporates the iterative philosophy of the prototype model, and adds iterative feedbacks from later iterations to previous iterations. Two examples of this type of model are the *spiral model* and the *incremental model.*

1.4.3.1 The Spiral Model

In contrast to the waterfall model, which is also called a document-driven model, the spiral model use risk as the criterion to terminate each iteration. Proposed by Beohm (1988), this model is called a risk-driven model. Figure 1.3 shows a picture of the spiral model. As the development proceeds, the software engineers move around the spiral in a clockwise direction. Cost and schedules are adjusted based on the risk analysis. This approach has its drawbacks. It is very difficult to manage the development processes under control. The success of this approach heavily depends on the success of the risk analysis. If a major risk is not uncovered and managed, problems are very likely occur.

1.4.3.2 The Incremental Model

Like the prototype model, the incremental model is also based on the waterfall model. Like the spiral model, the scope of the software is increased after each iteration. The incremental model applies the waterfall model to each iteration. Each linear sequence produces a deliverable increment of the software. When an incremental model is used, the first increment is often a *core product.* That

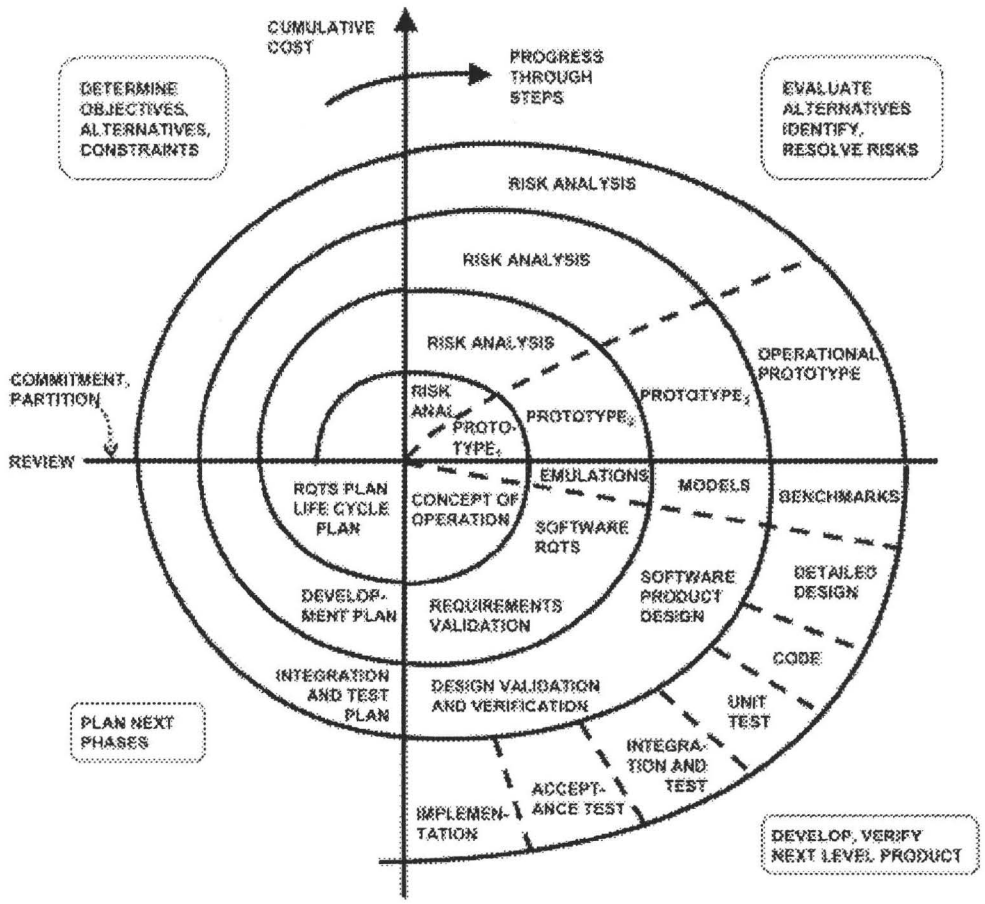

Figure 1.3: Spiral Model. Image from Beohm (1988)

is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered (Pressman, 1999). No prototypes are involved in this model. Each time the software is delivered as real software. The disadvantage of delivering a poor quality prototype is eliminated. A complete document is the criterion for terminating each stage in each iteration. With the information hiding principle (Parnas et al., 1984) in mind, the functionalities and corresponding documents may not need to change, or may change very little.

1.5 Proposed Methodologies for the Development of PMGT

While software development methodologies have been applied successfully for many applications, development of scientific computing software, including mesh generation software, still focuses primarily on the "programming" stage. PMGT is a library tool that will be called by other applications, such as FEA software. It is designed to be built on the Shared Hierarchical Academic Research Computing Network (SHARCNET), where SHARCNET is structured as a "cluster of cluster" designed to meet the computational needs of high performance computing researchers. PMGT is developed in this thesis to illustrate the use of software engineering methodologies to improve the quality of scientific computing software.

The quality factors that are important for PMGT include all quality factors for scientific computing software except for *Portability* (QF8), since PMGT is designed for a specific system. However, SHARCNET is constantly being improved. Hence, *Portability* (QF8) is desired if possible.

A scientific computing toolbox, like PMGT, has the following characteristics: *i*) embedded in another applications; *ii*) no direct interaction with the end users; and, *iii*) can start with a simple set of requirements and gradually add components. Given these characteristics, the incremental process model was chosen as the basis to developing PMGT. There are two iterations for the development of the software. The output of the first iteration does not involve parallelism. During the second iteration, some functionalities are im-

plemented using parallel algorithms. The process model used for developing PMGT is actually a modified incremental model. The modifications are as follows:

1. Feedback from later stages are added to previous stages in each iteration. That is, within each iteration, if significant problems due to decisions made during previous stages were discovered, then the decisions from the previous stages are modified before proceeding. The earlier a problem is found, the lower the cost to fix it. Adding feedback from later stages to previous stages can reduce the cost of a problem and thus improve the overall quality of the software, since the saved resource can be used for improving quality.

2. Commonality analysis was performed before the software requirements activities, where a commonality analysis is a process to study shared features or attributes among similar software to find possibilities for development of the software as a program family. The advantages of developing programs as a family are discussed in Dijkstra (1972) and Parnas (1976, 1978). The commonality analysis for mesh generation software has been discussed in Chen (2003); Smith and Chen (2004), and Cao (2006).

Four documents are generated during the development of PMGT, namely the Software Requirements Specification (SRS), the Module Guide (MG), the Module Interface Specification (MIS) and the Summary of Validation Testing Report (SVTR). These documents can improve the *Usability* (QF4) of PMGT

since they specify what PMGT do in different level of abstraction. Mathematical notation is used in the documents. It can improve the *Correctness* (QF1) and *Testability* (QF6) since it makes the SRS and the MIS unambiguous and validatable. Traceability matrices are also developed. These matrices can improve the *Correctness* (QF1), *Maintainability* (QF5), and *Flexibility* (QF7) of PMGT. The use relations of modules can improve the *Testability* (QF6) and *Reusability* (QF9), and the modification of algorithm can improve the *Efficiency* (QF3) and *Usability* (QF4). The automation of the correctness testing can improve the *Usability* (QF4) of PMGT.

The remainder of this thesis illustrates how software engineering methodologies are applied to the development of PMGT to improving the quality of the software. The organization of the rest of the thesis is as follows. Chapter 2 reviews the software requirements activities of PMGT. Chapter 3 outlines the architecture design of PMGT and gives more detail on the design of the software. Chapter 4 discusses implementation issues. Chapter 5 summarizes validation testing on PMGT. Chapter 6 provides conclusions from the thesis and addresses some extensions that can be studied in the future. In addition, the documentation for the SRS, MG, MIS, and SVTR are appended as appendices A, B, C, and D, respectively.

Chapter 2

Software Requirements

Although the qualities of software have been defined in the previous chapter, a metric is still needed to measure them. Like other engineering disciplines, software engineering should provide measurement to assess the quality of software. Software requirements can tell which of the qualities are most important and how to measure and evaluate whether the important qualities have been met. A software requirement is: *i*) a condition or capability needed by a user to solve a problem or achieve an objective; *ii*) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document; or, *iii*) a documented representation of a condition or capability as in the above two definitions (IEEE, 2000).

Software requirements can improve the following software qualities:

- Correctness (QF1): The process of writing software requirements, especially formal software requirements, helps the user understand what they

actually want to build.

- Usability (QF4): Software requirements provide the information of what the software can do. This information introduces the functionality of the software. The documentation of the functionality make the software easier to use.

- Maintainability (QF5): With the software requirements, maintainers can discover and locate errors by comparing the requirements with what the software actually does. Therefore, the software becomes easier to maintain.

- Testability (QF6): Software requirements serve as a contract between developers and testers. Without unambiguous and validatable software requirements, it is difficult to test the software.

- Reusability (QF9): The software can only be reused if what the software does is known. This information is easier to obtain by directly reading software requirements document than by deciphering the code.

Usually software requirements activities include software requirements elicitation, software requirements analysis, software requirements documentation, and software requirements verification. Software requirements elicitation facilitates the understanding of what the software is supposed to do. The software analysis is the process of refining and modeling the requirements. Software elicitation and software analysis are necessary for producing the software requirements document. The software requirement verification checks whether

the requirements are consistent and complete. In some case requirements verification is considered part of the analysis stage. However, it is separated in the current work to highlight the importance of verification step.

This chapter describes how these activities are conducted to develop PMGT. In Section 2.1, Section 2.2, Section 2.3, and Section 2.4, the activities of software requirements elicitation, software requirements analysis, software requirements documentation, and software requirements verification are specified, respectively. A complete SRS for PMGT is provided in Appendix A.

2.1 Software Requirements Elicitation

Requirements elicitation is the process of discovering the requirements for a system by communication with customers, system users and others who have a stake in the system development (Sommerville and Sawyer, 1997). The starting point for the current work was Smith and Chen (2004), which provides a set of software requirements that are common to mesh generation software. They also considered the differences between mesh generators in term of there variabilities. Smith and Chen (2004) significantly reduced the time and effort necessary to gather the requirements from stakeholders. However, the system analyzed by Smith and Chen (2004) was targeted at Finite Element Analysis (FEA) applications. PMGT, on the other hand, only manages the geometric information about the mesh, not other FEA related information, such as boundary conditions and material properties. Hence, only commonalities from Smith and Chen (2004) that are meaningful for PMGT were selected.

2.2 Software Requirements Analysis

Traditionally, requirements analysis methods are placed into two categories: structured analysis and object-oriented analysis. A limitation of both categories is that these methods associate requirements analysis with programming languages. Structured analysis relates to structured programming languages and object-oriented analysis relates to object-oriented programming languages. Focusing on specific class of programming languages brings the design decision to the early stage of the development. This invalidates the basic principles of software requirements that the software requirements should be abstract and methodology independent.

Goal-based methods are concerned with the use of goals for eliciting and analyzing requirements (van Lamsweerde, 2001). This kind of analysis is abstract since it does not favor any class of programming languages; hence, it gives more freedom for the later stage of development. PMGT used ideas from goal-based method for analyzing software requirements.

The first step of analysis is identifying the goals of PMGT. These goals are too general to be easily implemented as software, especially for the first iteration of the development. For example, a goal for PMGT may be to "refine a given mesh into a new mesh according to the provided information on which elements need to be refined." Without some restrictions (assumptions), such as the dimension of the input mesh, it would be impossible to develop the software. Hence, the goals need to be refined step by step to find requirements that are concrete enough to fit the scope of the software to be developed. Also, goals are usually expressed in natural language, and natural language

is inherently ambiguous. During the refinement, mathematical notations and terms are introduced to make the requirements validatable and unambiguous.

2.3 Software Requirements Documentation

A Software Requirements Specification (SRS) is a document containing a complete description of what the software will do, without describing how it will do it (Davis, 1990). According to their formality, methods for documenting requirements are categorized into informal methods, formal methods, and semi-formal methods. By using informal methods, software requirements are expressed in natural language. Most requirements documents in industry use informal methods due to the understandability of natural language. However, natural language is inherently ambiguous; hence, the requirements are difficult to validate. In contrast to informal methods, formal methods use languages designed for specification, such as Z, to document the requirements. The use of formal techniques can reduce the ambiguity of the requirements. However, this kind of method is not widely used due to understandability challenges and high cost. Semi-formal methods, such as UML, try to keep a balance between informal methods and formal methods. Semi-formal methods are easier to understand and develop than formal methods. However, problems of verification and validation of requirements still exist.

There is no universally accepted way of documenting requirements. A combination of informal methods and formal methods is used to document the software requirements of PMGT. For each requirement, plain English is used

for description. In addition, formal mathematical/logical expressions are used where applicable to improve the *Testability* (QF6) of PMGT. The format of the mathematical notations and terms are borrowed from Gries and Schneider (1993), as explained in Appendix A.

The template proposed by Lai (2004); Smith and Lai (2005); Smith et al. (2007) is used as a basis for the current SRS. Lai's template modified the general software requirements templates documented in IEEE (1998) and in Robertson and Robertson (2001). Lai's template was designed for the specific case of engineering mechanics software. Solving a engineering mechanics problem begins with generating theoretical models for the problem and then instantiating these models. The step of instantiating the model was removed for documenting PMGT, since PMGT is a general tool involving only a few mathematical equations. There is no need to instantiate the theoretical models, as the theoretical models already have the correct level of abstraction. Data constraints, which are part of Lai's template, are also not considered to be necessary in the current work. The complete SRS for PMGT is in Appendix A. Note that the SRS should be updated when the software requirements change. Sections of the SRS for PMGT are as follows:

1. Reference Material
2. Introduction
3. General System Description
4. Specific System Requirements
5. Other System Issues

6. Traceability Matrix

7. List of Possible Changes in the Requirements

8. Values of Auxiliary Constants

Each of these sections is described in more detail below, including examples where appropriate. Sections of *Reference Material, Introduction, General System Description,* and *Specific System Requirements* give introductions of PMGT in different perspectives, and improve the *Usability* (QF4) of PMGT.

2.3.1 Reference Material

This section includes tables of symbols, abbreviation and acronyms. These tables helps reduce the ambiguity of the document. For instance, a reader can refer to the table after reading the goal statements to see that M^{IN} and M^{OUT} are the symbols used for the input and output meshes, respectively. The reference material section also includes an index of requirements, to facilitate users quickly finding the requirements they need.

2.3.2 Introduction

This section gives an overview of the SRS for PMGT. First, the purpose of the documents is provided. Second, the scope of PMGT is identified. Third, some terminology about software engineering and mesh generation are defined. As mentioned in Section 1.3, most of mesh generation software is not developed by software engineers. Readers of the SRS for PMGT may not have essential knowledge of software engineering. Hence, including terminology about both

software engineering and mesh generation is necessary. Finally the organization of the document is summarized.

2.3.3 General System Description

This section describes the general information about the system. The interfaces between the system and its environment are defined first. Then the characteristics of potential users are discussed. At end of this section, some system constraints are described. This software is intended to be used on Shared Hierarchical Academic Research Computing Network (SHARCNET). However, SHARCNET is constantly improving and changing its system. Therefore, it is important to design the software to only need the basic features of SHARCNET, and not to focus on details of SHARCNET. Abstracting away the detailed system constraints makes it possible for PMGT to be used in other systems similar to SHARCNET, which improves the *Portability* (QF8) of PMGT.

2.3.4 Specific System Requirements

This section describes the system requirements in detail. After the problem is clearly and unambiguously stated, some solution characteristics are specified. Non-functional requirements are also included in this section. This section is the major section of the SRS. Goals, assumptions, theoretical models, data definitions, and software requirements are all specified here.

A goal is an objective that the system under consideration should achieve (van Lamsweerde, 2001). One of goals of PMGT is

G1: Given a mesh M^{IN} and instructions I on how to refine the mesh, PMGT should generate a refined mesh M^{OUT} according to the instructions I.

M^{IN} and M^{OUT} represent an input and an output mesh respectively, and I represents instructions on how a mesh should be refined/coarsened.

An assumption reduces the scope of the software. One example assumption of PMGT is

- **A1**: PMGT focuses on a 2D domain.
- **A4**: The input and output meshes are conformal.
- **A5**: The elements of input and output meshes are triangles.

Usually, the software is extended by relaxing one or more assumptions. Including assumptions in the SRS can make adding more functionality to the software easier by tracing from assumptions to requirements and then, in a later step, to modules.

Theoretical models refine the goals in two aspects. First, this refinement makes the goals more concrete by applying assumptions to the goals. Second, the refinement makes the goals more unambiguous by expressing theoretical models more formally. A theoretical model that refines G1 follows.

TM1: Refining Mesh

Input: M^{IN}: `MeshT`, I: `RCinstructionT`,

Output: M^{OUT}: `MeshT`

The following behavior is specified:

- *Refined*(M^{OUT}, M^{IN})

That is, the output mesh is a refined version of the input mesh.

Theoretical models, such as TM1, are more formal than the goals. However, they can be difficult to understand. Data definitions help by defining terms used in the theoretical models and requirements. Some of data definitions used to define TM1 are introduced as follows:

`RCinstrctionT` (D22) is defined as

> `RCinstructionT` := tuple of (*rORc*: `InstructionT`, *cInstr*: set of `CellInstructionT`)
>
> where `InstructionT` := {`REFINE, COARSEN, NOCHANGE`},
>
> and `CellInstructionT` := tuple of (*cell*: `CellT`, *instr*: `InstructionT`)

Refined(M^{OUT}, M^{IN}) (D23) is defined as

> *Refined*: `MeshT` × `MeshT` × `RCinstructionT` → $\mathbb{B}$
>
> *Refined*(m', m: `MeshT`, rc: `RCinstructionT`) ≡
>
> rc.rORc = `REFINE` ∧ *ValidMesh*(m) ∧ *ValidMesh*(m') ∧
>
> *CoveringUp*(m', m) ∧ # m' ≥ # m

All data definitions used in D23 are formally defined in the SRS. However, for the presentation in this chapter, it is not necessary to formally define all of the definitions. Some data definitions used to define D23 and data definitions used later in this section are explained informally as follows:

- `VertexT`: type of vertices;
- `EdgeT`: type of edges;

- `CellT`: type of cells;
- `MeshT`: type of meshes, which is a set of `CellT`;
- *Vertices*: a function that returns the vertices in a mesh;
- *Edges*: a function that returns the edges in a mesh;
- *BoundaryVertices*: a function that returns the set of boundary vertices;
- *BoundaryEdges*: a function that returns the set of boundary edges;
- *ValidCell*: a boolean function that returns true if the cell is a triangle;
- *Bounded*: a boolean function that returns true if the boundary edges form a closed polygon;
- *NoInteriorIntersect*: a boolean function that returns true if a point in space is inside only one cell of the mesh;
- *OnEdge*: a boolean function that returns true if a vertex is on the line segment between two vertices (exclusive) of an edge.

In the data definition D23, *ValidMesh* (D18) is a boolean function to check if a mesh is valid, which is defined as

ValidMesh: `MeshT` $\rightarrow \mathbb{B}$

ValidMesh(m: `MeshT`) $\equiv$ ($\forall$ e: `EdgeT` | e $\in$ *Edges*(m): *ValidEdge*(e))

$\wedge$ ($\forall$ c: `CellT` | c $\in$ m: *ValidCell*(c)) $\wedge$

Bounded(m) $\wedge$ *Conformal*(m)$\wedge$ *NoInteriorIntersect*(m)

CoveringUp (D19) is a boolean function to check if two meshes cover up one another, which is defined as

CoveringUp: `MeshT` $\times$ `MeshT` $\rightarrow \mathbb{B}$

CoveringUp(m1, m2: `MeshT`) $\equiv \forall$ v1, v2: `VertexT` |

v1 $\in$ *BoundaryVertice*(m1) $\wedge$ v2 $\in$ *BoundaryVertices*(m2):

($\exists$ b1, b2: `EdgeT` | b1 $\in$ *BoundaryEdges*(m1) $\wedge$

b2 $\in$ *BoundaryEdges*(m2):

(*OnEdge*(v1, b2) $\vee$ v1 $\in$ b2) $\wedge$ (*OnEdge*(v2, b1) $\vee$ v2 $\in$ b1))

Conformal (D16) is a a boolean function to check if a mesh is conformal, which is defined as

Conformal: `MeshT` $\rightarrow \mathbb{B}$

Conformal(m: `MeshT`) $\equiv \forall$ c1, c2: `CellT` | c1 $\in$ m $\wedge$ c2 $\in$ m

$\wedge$ c1 $\neq$ c2 :

($\exists$ e: `EdgeT` | e $\in$ *Edges*(m): ($\exists$ v: `VertexT` | v $\in$ *Vertices*(m):

(c1 $\cap$ c2 = e $\vee$ c1 $\cap$ c2 = v $\vee$ c1 $\cap$ c2 = $\emptyset$) $\wedge$ ($\neg$ *OnEdge*(v, e))))

The detailed data definitions can be found in the SRS.

The theoretical models can then be further refined to the functional requirements of the software. For each goal, assumption, theoretical model, data definition, and requirement, a name and a unique number are assigned for readability of the SRS, and for *Usability* (QF4) of PMGT.

All functional and nonfunctional requirements are specified in a tabular form. An example functional requirement is shown in Table 2.1. In each table, the field *Description* gives a brief description of this requirement. It

Requirements Number **Requirements Name**	F1 RefiningMesh
Description	PMGT should have capabilities for refining an existing mesh. I.rORc = `REFINE` $\wedge$ *Refined*($M^{\text{OUT}}, M^{\text{IN}}$)
Source	C1, V3
Related Data Definitions	D20, D22, D23
Related Theoretical Models	TM1
Binding Time	Scope time
History	Created - June, 2005. Modified - October, 2005. Change the name from "ImprovingMesh" to "RefiningMesh." Modified - October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" were added.

Table 2.1: An Example Functional Requirement

tells what PMGT should do to fulfill this requirement. There are two potential sources, shown in the *Source* field, for each requirement. One source is from Smith and Chen (2004), and the other comes from Dr. Smith. If the requirement is from Smith and Chen (2004), then this field will show the commonality number, with a prefix C and the associated variability, shown by a prefix V. Where applicable, *Related Data Definitions* and *Related Theoretical Models* give the numbers of the related data definitions and the numbers of the related theoretical models, respectively. These two field only appear for functional requirements. The *Binding Time* field either shows scope time or run time. *Scope time* means that this requirement is determined when the SRS is written. *Run time* means that this requirement is determined when

the system is running. *History* records the time and details of creating and changing the requirement.

An example nonfunctional requirement (NFR) is shown in Table 2.2. An NFR has similar fields to a functional requirement. However, there is no

Requirements Number **Requirements Name**	N1 Performance
Description	Refining/coarsening a mesh using multiple processors should be faster than when using a single processor. In addition, the performance of PMGT should be comparable with that of similar applications. The execution time to refine an example mesh, which is specified in Appendix D, should be *RSPTIME*.
Source	C15, V39
Binding Time	Scope time
History	Created – June, 2005.

Table 2.2: An Example Nonfunctional Requirement

Related Data Definitions and *Related Theoretical Models*, since there are no such relations. As mentioned Section 1.2, *Nonfunctional Requirement Challenge* exists in scientific computing software, including PMGT. NFRs are difficult to formally document and test. In the case of PMGT, an attempt was make to give a criterion for each NFR so that it is possible to test whether the requirement has been met. For example, in the *Description* of the nonfunctional requirement N1, the criterion is quantified by giving a constant *RSPTIME*, which will be specified in Section 2.3.8. The quantifying of the requirements improves the *Testability* (QF6) of PMGT.

2.3.5 Other System Issues

This section includes some other supporting information that might contribute to the success or failure of the system development. Open issues, off-the-shell solutions, and waiting room items are considered here. In particular, the waiting room items relate to relaxing the assumptions introduced in Section 2.3.4. The waiting room provides a blueprint of how the system will be extended, and hence it improves the *Flexibility* (QF7) of the software.

2.3.6 Traceability Matrix

This section shows the traceability matrix. This matrix gives the associations among goals, assumptions, data definitions, theoretical models, and functional and nonfunctional requirements. A portion of the matrix is shown in Table 2.3 and Table 2.4. This matrix can be used for improving the *Maintainability* (QF5), and *Flexibility* (QF7) of PMGT. For example, if one of the goals of PMGT changes, all of the assumptions, most of data definitions, one of the theoretical models, and most of functional requirements would change. On the other hand, if the assumption that "the input and output meshes are conformal" (A4) changes, only the data definition D16 and the requirement F7, which relate to comformality , would change. Another use of the matrix is to improve the *Correctness* (QF1) of PMGT since the matrix ensures that the initial goals are correctly transferred into the software requirements. Other uses of the traceability matrices are specified in Section 2.4.

2.3.7 List of Possible Changes in the Requirements

The system might evolve to accommodate some changes in the future. These changes will add additional goals to the software library. For example, the input of PMGT may change to include material properties. Including this information improves the *Flexibility* (QF7) of PMGT.

2.3.8 Values of Auxiliary Constants

The constants given in this section are used to make some of the nonfunctional requirements validatable. These constant are defined to quantify the nonfunctional requirements by comparing them to a similar software product, such as AOMD (SCOREC, Last Access: January, 2006). For example *RSPTIME* is defined as the execution time to refine the same mesh as that specified in nonfunctional requirement N1 using AOMD. It is noticed that these constants are for specific system. As mentioned previously, this quantifiable requirement improves the *Testability* (QF6) of PMGT.

2.4 Software Requirements Verification

An important part of the requirement analysis is verifying the requirements for completeness and consistency. A traceability matrix can helps with this activity. For example, the traceability matrix checks for completeness since if there is no check mark (✓) in a cell associated with a goal, or a assumption, or a theoretical model in the corresponding column, it means the goal or the assumption, or the theoretical model is not address by any software require-

ment. Hence, the software requirements are not complete. The traceability matrix can also partially check the consistency of the requirements document. If there are no entries in the column associated with a data definition, it means the data definition is not useful and should not appear in the document. If there are no data definitions in columns associated with a theoretical model in a row, this model should be checked carefully to see if there are any potential deficiencies in the software requirements, potentially due to software requirements analysis problems.

	G1	G2	A1	A2	A3	A4	A5	A6	TM1	TM2
A1	✓	✓	✓							
A2	✓	✓		✓						
A3	✓	✓			✓					
A4	✓	✓				✓				
A5	✓	✓	✓				✓			
A6	✓	✓						✓		
D1	✓	✓	✓							
D2	✓	✓								
D3	✓	✓								
D4	✓	✓	✓				✓			
D5	✓	✓	✓				✓			
D6	✓	✓	✓				✓			
D7	✓	✓								
D8	✓	✓	✓							
D9	✓	✓	✓							
D10	✓	✓	✓				✓			
D11	✓	✓								
D12	✓	✓	✓					✓		
D13	✓	✓	✓							
D14	✓	✓	✓							
D15	✓	✓	✓	✓						
D16	✓	✓	✓			✓				
D17	✓	✓	✓	✓						
D18	✓	✓								
D20	✓	✓								
D21	✓	✓								
D22	✓	✓								
D19	✓	✓	✓	✓						
D23	✓									
D24		✓								
TM1	✓		✓						✓	
TM2		✓	✓							✓

Table 2.3: Traceability Matrix (PART I): Goals, Assumptions, Theoretical Models, Data Definitions, and Requirements (I)

	G1	G2	A1	A2	A3	A4	A5	A6	TM1	TM2
F1	✓								✓	
F2		✓								✓
F3	✓	✓							✓	✓
F4					✓					
F5	✓	✓	✓				✓		✓	✓
F6	✓	✓	✓						✓	✓
F7	✓	✓				✓				
F8	✓	✓							✓	✓
F9	✓	✓							✓	✓
F10	✓	✓								
F16	✓	✓								

Table 2.4: Traceability Matrix (PART I): Goals, Assumptions, Theoretical Models, Data Definitions, and Requirements (II)

Chapter 3

Design

According to the process model for PMGT proposed in Section 1.5, once software requirements have been analyzed and specified, software design follows. Software design is defined as decomposing the software into modules, describing what each module is intended to do and specifying the relationship among the modules (Ghezzi et al., 2003). A module is defined as a "work assignment," as proposed by Parnas (1972). Instead of "a portion of a program," this definition includes in software design the activities that occur before programming. Because the module decomposition of PMGT does not assume any programming language, a different class of programming language will potentially change the implementation. The independence of PMGT from the specific programming language means that the *Reusability* (QF9) of the design is improved.

The principle applied for design is *information hiding*. According to this principle, system details that are likely to change independently should

be hidden in different modules (Parnas et al., 1984). The information hiding principle allows both designers and maintainers to easily identify the parts of the software that they want to consider without needing to know irrelevant details. The *Maintainability* (QF5) and *Flexibility* (QF7) of the software are thus improved. The process of module decomposition can proceed in different ways, such as top-down or bottom-up. The top-down process decomposes the system by stepwise refinement from higher levels of abstraction to lower levels of abstraction. In contrast, bottom-up decomposition first defines modules and then iteratively combines these modules into higher level components. Like the design of most software, the strategy used here is a combination of top-down and bottom-up design.

Applying the top-down strategy, the whole design of PMGT is decomposed into an architectural design, which will be specified in Section 3.1, and a detailed design, which will be introduced in Section 3.2.

3.1 Architectural Design

Software architecture is the overall structure of the software and the ways in which that structure provides conceptual integrity for a system (Shaw and Garlan, 1995). The overall structure is illustrated in Section 3.1.1. The completeness and consistency of this structure are verified in Section 3.1.2. The use relation is shown in Section 3.1.3. The architectural design is documented in the MG, which is appended in Appendix B.

3.1.1 Decomposition of the System into Modules

By hiding details that are likely to change independently in different modules, PMGT is easier to maintain and extend. For example, if one of algorithms that refines the input mesh changes, only the modules that hide the information on how to refine a mesh need to change. It is not likely that the modules that hide the information on how to coarsen a mesh need to change. This *design for change* approach is adopted throughout the architectural design of PMGT.

There are two steps for designing the architecture of PMGT. In the first step, anticipated changes are identified. These changes should not impact the basic functionality of PMGT in that the goals of the software should not be affected. The list of anticipated changes can improve the *Flexibility* (QF7) of PMGT since this list helps to find modules to be changed when software requirements change. Several examples of anticipated changes (from Appendix B) are as follows:

- AC7: The mechanisms for validating the input and output meshes.
- AC11: The data structure of a mesh.
- AC12: The algorithms for refining a mesh.
- AC14: The shape of a cell, which is initially assumed to be a triangle.

Ideally, all anticipated changes should be independent of one another, so that one change can be hidden inside one module. When a change occurs, only the module that hides the change needs to be modified.

Unlikely changes are also listed. If one of these changes occurs, the design of PMGT makes no obligation that adapting to this change will only require small modifications. Some unlikely changes include the following:

- UC5: The type of the mesh is unstructured.
- UC6: The representation of an edge as a set of vertices.
- UC8: A Cartesian coordinate system is used.

An unstructured mesh is more complex than a structured one. Therefore, it is not likely to adapt unstructured mesh software to structured mesh software because of the greater generality of the unstructured mesh. Hence, the type of unstructured mesh is an unlikely change (UC5). Another example of an unlikely change is the change of the representation of an edge (UC6). This change will affect all of the data structure modules. Unlike the design proposed by ElSheikh et al. (2004), the coordinate system is also assumed to be an unlikely change (UC8). Putting UC8 into the anticipated changes category would make the software more general. However, this generality would be at the price of complexity. A design decision was made to consider the UC8 to be an unlikely change because this decision reduces the complexity of PMGT. Listing unlikely changes helps one set realistic goals for *Flexibility* (QF7) because it explicitly identifies those maintenance tasks that would not be likely nor feasible.

After identifying likely and unlikely changes, the system was decomposed into modules. A bottom-up strategy was used for the decomposition to facilitate the principle of information hiding. Each module accommodates one

(or more) anticipated changes. These modules were iteratively combined to form higher level modules until the whole system was constructed, as shown in Table 3.1. The level 1 decomposition into hardware-hiding module, behavior-hiding module and software decision module was inspired by Parnas et al. (1984).

Level 1	**Level 2**	**Level 3**	**Level 4**
Hardware-Hiding Module	Extended Computer Module	Virtual Memory Module	
		File Read/Write Module	
	Device Interface Module	Keyboard Input Module	
		Screen Display Module	
Behavior-Hiding Module	Input Format Module		
	Output Format Module		
	Service Module		
Software Decision Module	Mesh Data Module	Entity Module	Vertex Module
			Edge Module
			Cell Module
		Mesh Module	
	Algorithm Module	Refining Module	
		Coarsening Module	

Table 3.1: Module Hierarchy

Each module has its secrets and provides services to the other modules. Only the leaves in the hierarchy have to be implemented. The higher level modules are conceptual. They are used to facilitate reading of the MG for understanding the design. Some of the leaf modules, such as the leaves in the "Hardware Hiding Modules," are commonly used in many different software

projects. These module are usually implemented by the operating system or through the libraries of the implementing programming language.

In the MG, in addition to *Secrets* and *Services*, there is an *Implemented By* field for each module. If the entry in this field is *OS*, then this module is assumed to already be implemented. *PMGT* means this module will be implemented by PMGT. If a dash (-) is shown, this means that this module does not need to be implemented. Whether this module is implemented depends on the programming language used. For example, if an imperative programming language is used, the higher level modules will not likely be implemented. However, if inheritance exists in the implementing programming language, such as in an OO language, the higher level modules can be implemented as super classes. As mentioned previously, the decomposition of PMGT is independent of programming language. Examples of module decomposition are illustrated as follows:

- **M4: Screen Display Module**

  Secrets: The data structure and algorithms to display graphics and text on the screen.

  Services: Provides an interface between the system and the screen so the system can display information on the screen through the use of programs in the module.

  Implemented By: OS

- **Behavior-Hiding Module**

  Secrets: The contents of the required behaviors.

Services: Includes programs that provide externally visible behavior of the system as specified in the software requirements specification (SRS) documents. This module serves as a communication layer between the hardware-hiding module and the software decision module. The programs in this module will need to change if there are changes in the SRS.

Implemented By: –

- **M12: Refining Module**

  Secrets: Algorithms for refining a mesh.

  Services: Refining a mesh.

  Implemented By: PMGT

3.1.2 Verifying the Decomposition

The decomposition can be verified for consistency and completeness by considering the traceability matrices. The traceability matrix between modules and requirements is shown in Table 3.2. *M* followed by a number is the number of a module. *F* followed by a number is a number of a functional requirement. *N* followed by a number is the number of a nonfunctional requirement. There is also a special column "Doc," which represents the documentation of PMGT. No empty row in the table means that all requirements of PMGT are fulfilled by one or more modules; that is, the design is complete. No empty column means that all modules are necessary to implement one or more requirements; that is, the design is consistent.

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	Doc
F1												✓		
F2													✓	
F3												✓	✓	
F4									✓	✓	✓	✓	✓	
F5							✓			✓				
F6								✓				✓	✓	
F7							✓					✓	✓	
F8		✓			✓									
F9												✓	✓	
F10	✓	✓				✓								
F11						✓								
F12						✓								
F13						✓								
F14						✓								
F15						✓								
F16														✓
N1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
N2					✓	✓	✓	✓	✓	✓	✓	✓	✓	
N3	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	
N4	✓	✓	✓	✓	✓	✓	✓							
N5					✓	✓	✓	✓	✓	✓	✓	✓	✓	
N6			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
N7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 3.2: Traceability Matrix: Modules and Requirements

The traceability matrix improves *Maintainability* (QF5) and *Flexibility* (QF7). For example, if the requirement that "PMGT should have capabilities for refining an existing mesh" (F1) changes, then only the "Refining Module" (M12) would need to change since from the traceability matrix only M12 is associated with F1. Another example is that if the requirement that the shape of the elements in a mesh is triangular (F5) changes, then only the service module (M7), which hides the information on how to validate a mesh, and the

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13
AC1	✓												
AC2		✓											
AC3			✓										
AC4				✓									
AC5					✓								
AC6						✓							
AC7							✓						
AC8								✓					
AC9									✓				
AC10										✓			
AC11											✓		
AC12												✓	
AC13													✓
AC14							✓			✓			

Table 3.3: Traceability Matrix: Modules and Anticipated Changes

cell module (M10), which hides the information on the data structure of a cell, would need to change. The matrix also can improve the *Correctness* (QF1) of PMGT since the matrix ensures that the software requirements are correctly transferred into the modules.

The matrix in the functional requirements area is sparse; that is, changing one of functional requirements will not change too many modules. On the other hand, changing one of the nonfunctional requirements means changing many modules. This demonstrates that nonfunctional requirements are associated with qualities of the system, not specific functions.

Another traceability matrix is the matrix between modules and anticipated changes, as shown in Table 3.3. As in Table 3.2, *M* followed by the number is the number of a module. *AC* followed by a number is a number of an anticipated change. Except for AC14 (anticipated change for the shape

of cells), M7 (service module), and M10 (cell module), which have multiple associations, the rest of the anticipated changes and modules have single associations. For each module, there is only one anticipated change associated with it; that is, the module is simple since it only have one secret (anticipated change). For each anticipated change, there is usually only one module associated with is; that is, the module is as independent as possible, since each secret is only hidden in one module. Changing one anticipated change usually only requires changing one module. For example, if the data structure of a mesh (AC11) changes, then the mesh module (M11) would change, since from Table 3.3 AC11 associates with M11. Decomposing PMGT to simple and independent modules promotes *Testability* (QF6), *Maintainability* (QF5) and *Flexibility* (QF7).

3.1.3 Use Relation

Software design includes relationship among modules. The use relation for PMGT is shown in Figure 3.1. Parnas (1978) said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system. This improves the *Testability* (QF6) and *Reusability* (QF9) of PMGT. For example, the mesh module (together with vertex module, edge module, and cell module) is a subset of the system. The design and the implementation

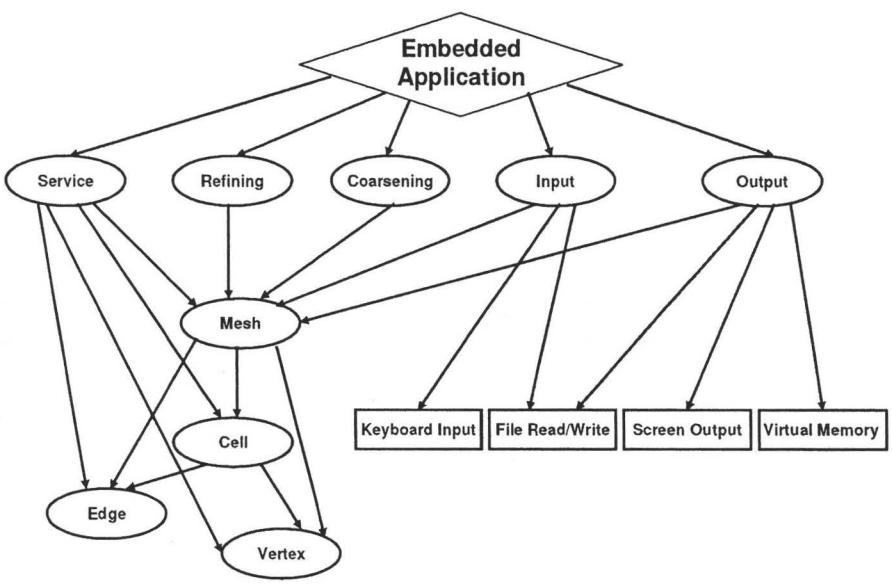

Figure 3.1: Uses Hierarchy among Modules

of this subset can be reused since it does not use other modules. Another example is that the coarsening module and output format module can be removed, and the remaining subset is still very useful. Modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels, thus the *Maintainability* (QF5) of PMGT is improved. The design is easily understood due to its simplicity; therefore, the *Usability* (QF4) of PMGT is improved.

3.2 Detailed Design

Figure 3.1 gives the uses relations between modules. However, these relations do not give enough information for each module to be developed independently. The syntax and semantics of the access routines for each module are still needed. The detailed design of PMGT being described in this section provides this information by specifying the interface of each module. A document that provides the detailed design, called the Module Interface Specification (MIS), is appended in Appendix C. The detailed design is less abstract than the architectural design in the last section. However, it is still abstract because it describes what the module will do, but not how to do it. A state machine MIS is used. Note that some of the modules have multiple projections, as used for example in Bauer (1995). In this case, state variables give the format of all states for all of the created objects. The change of state variables is applicable to the particular object associated with this module. Before giving some examples to illustrate the detailed design of PMGT in Section 3.2.2,

the template for documenting the MIS for each module is first introduced in Section 3.2.1.

3.2.1 Template

The template used to document the MIS for each module is a modified version of the MIS template presented in Ghezzi et al. (2003) and of that presented in Hoffman and Strooper (1999). According to the adopted template, each module is modeled as a finite state machine. It has a set of state variables, inputs, outputs, and transitions. In the case that some conditions do not hold, an exception is raised by the access program that detects the exception. If an access program has an output, then *Output* is specified. If an access program changes states variables, a *Transition* is specified. The inputs of the access program are listed as arguments. The mathematical notations used in the MIS follow that introduced by Gries and Schneider (1993), as illustrated in the SRS document in Appendix A. The whole template is composed of four parts as follows:

1. *Module Name*: This section gives the name of the module.

2. *Uses*: This section lists constants, data types, and access programs that are defined outside of this module. The format of each imported item is specified as

   Uses ⟨ *module name* ⟩ **Imports** ⟨ *resource constants/data_type/access_program list* ⟩

   The associations with other modules are listed when the use of the other

modules is necessary to document the MIS. It is not necessarily the same as the uses relation in Section 3.1.3. For example, in the examples listed in Section 3.2.2, the refining modules uses *isValidMesh* and *CoveringUp* functions defined in the Service module to specifying the assumptions and semantics of the access program. However, the Service module is not used to fulfill the functionality of the refining module. Hence, there is no uses relation between the refining module and the service module. On the other hand, it is obvious that the refining module will use the *addCell* function to refine a mesh. However, this function is not used by the refining module for specifying the semantics of the access program of the refining module. Hence, the *addCell* function is not imported by the refining module in the MIS document.

3. *Interface Syntax*: This section defines the syntax of the module interface. The interface indicates the services that the module provides. Other modules can only access this module through this interface. The other information inside the module is the secret that it hides from other modules. Changing this internal information will not affect the way that other modules use this module. This section includes the exported constants, exported data types, and exported access programs. Each access program has a name, input list, output list, and exceptions.

4. *Interface Semantics*: This section introduces the semantics associated with the above syntax. It includes *i*) state variable; *ii*) assumption; *iii*) access program semantics; *iv*) local functions; *v*) local data types;

vi) local constants; and, *vii*) considerations. The access program semantics include possible exceptions, possible outputs, and possible transitions. The semantics should be as formal as possible to improve the *Testability* (QF6) of PMGT. When necessary and appropriate, an English explanation is included to help readers understand the meaning of the mathematical notation. The natural language explanation improves the *Usability* (QF4) of PMGT. Both assumptions in the interface semantics and exceptions in the access program semantics specify abnormal situations. However, exceptions and assumptions serve different purposes. When an assumption does not hold, the software makes no guarantee on the behavior. On the other hand, when an exception occurs, the software is obligated to handle it. Local functions, local data types, and local constants are used to facilitate the expression of the interface semantics. The considerations section includes other issues related to the MIS of this module that could not be covered in the other parts.

3.2.2 Examples

The detailed design of each module is documented according to the template in Section 3.2.1. As shown in Table 3.1, the system is decomposed into three modules in Level 1. Hardware-Hiding Module is implemented outside of PMGT and not included in the examples. Service Module, which belongs to Behavior-Hiding Module is selected as one of examples. Software Decision Module is the most important module. Hence, two example modules that belong to the Software Decision Module are illustrated here. One is the Mesh Module, which

is for the data structure of the mesh, and the other is Refining Module, which is for one of the algorithms. These three examples that illustrate the idea of the detailed design are provided in the section that follows. If an MIS subsection does not have content, then this section is excluded to save space. In addition, not all of the semantics of the access programs are listed to simplify the presentation. The full details for these example can be found in Appendix C.

3.2.2.1 Mesh Module

- Imported Data Types:

  Uses *Vertex Module* **Imports** `VertexT`

  Uses *Edge Module* **Imports** `EdgeT`

  Uses *Cell Module* **Imports** `CellT`

- Exported Data Types: `MeshT` := set of `CellT`

- Exported Access Programs: The exported access programs for the mesh module are listed in Table 3.4.

- State Variables: m: set of `CellT`

- Invariant: $\#m \geq 0$

- Assumptions: initMesh() is called before any other access routines.

- Access Program Semantics:

  - initMesh()

Routine Name	Input	Output	Exceptions
initMesh			
getMesh		MeshT	
numOfCells		$\mathbb{N}$	
addCell	CellT	MeshT	CellExist
deleteCell	CellT		CellNotExist
onEdge	VertexT, EdgeT	$\mathbb{B}$	
belongToCell	EdgeT , CellT	$\mathbb{B}$	
inside	VertexT , CellT	$\mathbb{B}$	
vertices		set of VertexT	
edges		set of EdgeT	
boundaryEdges		set of EdgeT	
boundaryVertices		set of VertexT	

Table 3.4: Exported Access Programs of the Mesh Module

* **Transition**

  $m := \emptyset$

– addCell(c: CellT)

* **Exception**

  $c \in m \Longrightarrow$ CellExist

* **Transition**

  $m := m \cup \{c\}$

– onEdge(v: VertexT, e: EdgeT)

* **Description**

  Returns true if a vertex v is on the line segment between two vertices (exclusive) of the edge e.

* **Output**

  $\exists\, v1, v2$: VertexT |

  $v1 \in e \wedge v2 \in e \wedge v1 \neq v2 \wedge v \neq v1 \wedge v \neq v2$:

$(v1.x < v.x \leq v2.x \wedge$

$(v.y - v1.y)/(v.x - v1.x) = (v2.y - v1.y)/(v2.x - v1.x))$

- belongToCell(e: `EdgeT`, c: *CellT*)
  - **Description**

    Returns true if an edge e belongs to a cell c.
  - **Output**

    $\forall\ v$: `VertexT` $\mid v \in e : v \in c$
- edges()
  - **Description**

    Returns a set of all edges of the mesh
  - **Output**

    $\{v1, v2 :$ `VertexT` $\mid (\forall c$: `CellT` $\mid c \in m :$

    $v1 \in c \wedge v2 \in c \wedge v1 \neq v2) : \{v1, v2\}\}$
- boundaryEdges()
  - **Description**

    Returns the set of boundary edges of the mesh
  - **Output**

    $\{b$: `EdgeT` $\mid b \in$ edges() $\wedge$

    $(\#\{c$: `CellT` $\mid c \in m \wedge$ belongToCell(b, c): $c\} = 1)$: $b\}$

3.2.2.2 Service Module

- Imported Data Types:

  Uses *Vertex Module* **Imports** `VertexT`

Uses *Edge Module* **Imports** `EdgeT`

Uses *Cell Module* **Imports** `CellT`

Uses *Mesh Module* **Imports** `MeshT`

- Imported Access Programs:

  Uses *Mesh Module* **Imports** onEdge(), inside(),

  vertices(), edges(), boundaryEdges(), boundaryVertices()

- Exported Data Types:

  `InstructionT` := {`REFINE, COARSEN, NOCHANGE`}

  `CellInstructionT` := tuple of (*cell*: `CellT`, *instr*: `InstructionT`)

  `RCinstructionT` := tuple of

  (*rORc*: `InstructionT`, *cInstru*: set of `CellInstructionT`)

- Exported Access Programs:

  The exported access programs for the services module are listed in Table 3.5.

Routine Name	Input	Output	Exceptions
isValidMesh	`MeshT`	$\mathbb{B}$	
coveringUp	`MeshT` $\times$ `MeshT`	$\mathbb{B}$	

Table 3.5: Exported Access Programs of the Service Module

- Access Program Semantics

  - isValidMesh(*m*: `MeshT`)

    * **Description**

      Returns true if the cells of the mesh are bounded, conformal, and if any two cells are not overlapping.

* **Output**

  Bounded(m) $\wedge$ Conformal(m) $\wedge$ NoInteriorIntersect(m)

– coveringUp($m1$: `MeshT`, $m2$: `MeshT`)

* **Description**

  Returns false if any boundary vertex of one mesh is not on a boundary edge of another mesh. Otherwise, return true.

* **Output**

  $\forall v1, v2$: `VertexT`, |

  $v1 \in$ boundaryVertices($m1$) $\wedge$ $v2 \in$ boundaryVertices($m2$):

  ($\exists$ $b1$, $b2$: `EdgeT` | $b1 \in$ boundaryEdges($m1$) $\wedge$

  $b2 \in$ boundaryEdges($m2$):

  (onEdge($v1$, $b2$) $\vee$ $v1 \in b2$) $\wedge$ (onEdge($v2$, $b1$) $\vee$ $v2 \in b1$))

• Local Functions

– ValidCell: `CellT` $\rightarrow \mathbb{B}$

  ValidCell(c: `CellT`) $\equiv \#c = 3 \wedge$ Area(c) ≥ 0

– Bounded: `MeshT` $\rightarrow \mathbb{B}$

  Bounded(m: `MeshT`) $\equiv \forall$ v: `VertexT` | $v \in$ boundaryVertices(m):

  ($\#\{e$: `EdgeT` | $e \in$ boundaryEdge(m) $\wedge$ $v \in e : e\} = 2$)

– Conformal: `MeshT` $\rightarrow \mathbb{B}$

  Conformal(m: `MeshT`) $\equiv \forall$ $c1$, $c2$: `CellT` |

  $c1 \in m \wedge$ $c2 \in m \wedge c1 \neq c2$:

  ($\exists$ e: `EdgeT` | $e \in$ eges(m): ($\exists$ v: `VertexT` | $v \in$ vertices(m):

  ($c1 \cap c2 = e \vee$ $c1 \cap c2 = v \vee$ $c1 \cap c2 = \emptyset$) $\wedge$ ($\neg$ onEdge(v, e))))

- NoInteriorIntersect: `MeshT` $\rightarrow \mathbb{B}$

  NoInteriorIntersect(m: `MeshT`) $\equiv \forall\ c1, c2 :$ `CellT` $|$

  $c1 \in m \wedge\ c2 \in m \wedge\ c1 \neq c2 :$

  $(\forall\ v :$ `VertexT` $|$ inside(v, $c1$): $\neg$ inside(v, $c2$))

3.2.2.3 Refining Module

- Imported Data Types:

  Uses *Mesh Module* **Imports** `MeshT`

  Uses *Service Module* **Imports**

  `InstructionT, CellInstructionT, RCinstructionT`

- Imported Access Programs:

  Uses *Service Module* **Imports** isValidMesh(), coveringUp()

- Exported Access Programs: The exported access programs for vertex module is listed in Table 3.6.

Routine Name	Input	Output	Exceptions
refining	`MeshT` $\times$ `RCinstructionT`	`MeshT`	

Table 3.6: Exported Access Programs of the Refining Module

- Assumptions: isValidMesh(m) and $i.rORc =$ `REFINE`

  for input m: `MeshT` and i :`RCinstructionT`

- Access Program Semantics:

  - refining(m: `MeshT`, i: `RCinstructionT`)

* **Output**

  m'

  such that

  ValidMesh(m) $\wedge$ ValidMesh(m') $\wedge$ CoveringUp(m', m) $\wedge$

  $\#m' \geq \#m$

Chapter 4

Implementation

The system implementation is the transformation of the design to a work product. The implementation phase is very important in the software development life cycle because it produces an executable version of the system. Most of the quality factors mentioned in Section 1.1 are reflected through this work product.

Unlike other phases in the software development life cycle, which are very simple or completely missing in scientific computing software, the implementation is always part of scientific computing software. However, even when the implementation is the sole component, it is not always done well. One of reasons for poor quality scientific computing software is that most scientific computing software, including mesh generation software, is written by scientist and most scientists have simply never been shown how to program efficiently (Wilson, 2006).

The implementation is the final step of the refinement from abstract to

concrete. The major decisions relating to the implementation of PMGT are the data structure, algorithms, and programming language. A considerable effort has been spent on studying the data structures and algorithms for mesh generation. This thesis is not intended to develop a brand new data structure or algorithm for mesh generation. Instead, considerations are given to choose (with minor modifications if necessary) proper data structure and algorithms to fit the scope of PMGT and to improve the qualities of PMGT. The selection of the data structure, the algorithms, and the programming language for the serial version of PMGT are discussed in Section 4.1, Section 4.2, and Section 4.3, respectively. Other decisions related to the implementation of PMGT, such as the decisions about parallelism and the system, are discussed in Section 4.4. This chapter also includes an introduction to the software technologies used to improve the quality of PMGT in Section 4.5.

4.1 The Data Structures

A mesh can be represented by a list of cells, and each cell can be represented by a list of vertices. This data structure that is used for representing meshes in the previous chapters is simple and easy to understand (with meshes as sets of cells, cells as sets of vertices and vertices as tuples of real numbers). It is a good choice for representing a mesh during the software requirements and design stages since they are abstract and understandability is of high importance. However, in practice, this data structure is too inefficient. Although the geometrical information (the positions of the vertices) is given, the topological

information needs to be more detailed. Whenever information, like what cells are adjacent to a particular cell, is needed, searching the entire list will be necessary. On the other hand, if all geometrical and topological information are stored, too much space will be required. A compromise must be made to keep a balance between understandability, which relates to *Maintainability* (QF5) and *Flexibility* (QF7), and *Efficiency* (QF3). In this section potential mesh data structures are surveyed in Section 4.1.1. The data structure that PMGT uses is illustrated in Section 4.1.2.

4.1.1 The Current Approach

Berti (2000) names two kinds of relations among mesh entities: incidence relations and adjacency relations, which are widely used in the mesh generation community. This thesis adopts this naming convention. An incidence relation is a relation between the different classes of mesh entities, such as a relation between a cell and one of its edges. The edge is called an incident edge of the cell, and the cell is called the incident cell of the edge. An adjacency relation is a relation between the same class of mesh entities, such as two cells. For instance, one cell can be adjacent to another cell.

The data structures of a polygonal mesh, in which the shape of cells is a polygon, are mainly divided into two categories, face-based data structures and edge-based data structures. In two dimensional space, a face is a cell. For each kind of data structure, geometric information is stored; that is, a list of vertices is stored. The difference comes from what and how the topological information is stored; that is, what and how the incidence relation and adjacency relation

is stored.

Face-based data structures store, for each face, the incident vertices, and its neighboring faces. Navigating around each vertex can be made by visiting all surrounding faces. The "Triangle" mesh generation software (Shewchuk, Last Access: January, 2006) uses a face-based data structure. Face-based data structures are more efficient for a mesh in which the shape of faces is the same due to the use of an array to store the adjacent faces. However, it is not efficient for a mixed mesh, in which the shape of cells may vary, since the number of adjacent faces may vary.

Edge-based data structures store an incident vertex, an incident face and its neighboring edges for each edge. A good example of edge-based data structures is the *halfedge* data structure, as illustrated in Figure 4.1. The halfedge data structure has its name because instead of storing the edges of the mesh, halfedges are stored. As the name implies, a halfedge is a half of an edge and is constructed by splitting an edge down its length. The two half-edges make up an edge pair. Half-edges are directed. A halfedge is called an outgoing halfedge of a vertex if the vertex is the starting point of the edge. On the other hand, a halfedge is called an incoming halfedge of a vertex if the vertex is the target point of the edge. The two halfedges of a pair have opposite directions, and each halfedge is called the opposite halfedge of the other. Different halfedge data structures vary in some minor details. The halfedge data structure used by OpenMesh (Last Access: January, 2006) is illustrated in Figure 4.1. The numbers refer to the following (where $\mapsto$ means *h*as an attribute of):

1. Vertex $\mapsto$ one outgoing halfedge
2. Face $\mapsto$ one halfedge
3. Halfedge $\mapsto$ target vertex
4. Halfedge $\mapsto$ its face
5. Halfedge $\mapsto$ next halfedge
6. Halfedge $\mapsto$ opposite halfedge (implicit)
7. Halfedge $\mapsto$ previous halfedge (optional)

As shown by Figure 4.1, in this data structure, each vertex stores one outgoing halfedge's information. There is more than one outgoing halfedge for a vertex. It does not matter which outgoing halfedge is stored since searching for adjacent outgoing edges for an edge can be done in both a clockwise and counterclockwise order. Each face stores one incident halfedge of the face. Again this incident halfedge can be any halfedge that is incident to the face since a search can be performed on halfedges that are incident to the face.

In the edge-based data structure the size of the storage is fixed. Therefore, it is more efficient for a mixed mesh than a face-based data structure, since a fixed size array may be used.

4.1.2 The Data Structure for PMGT

Although PMGT can only deal with triangular mesh, it is likely to be extended to accommodate quadrilateral mesh since 90% of the FEA applications deal

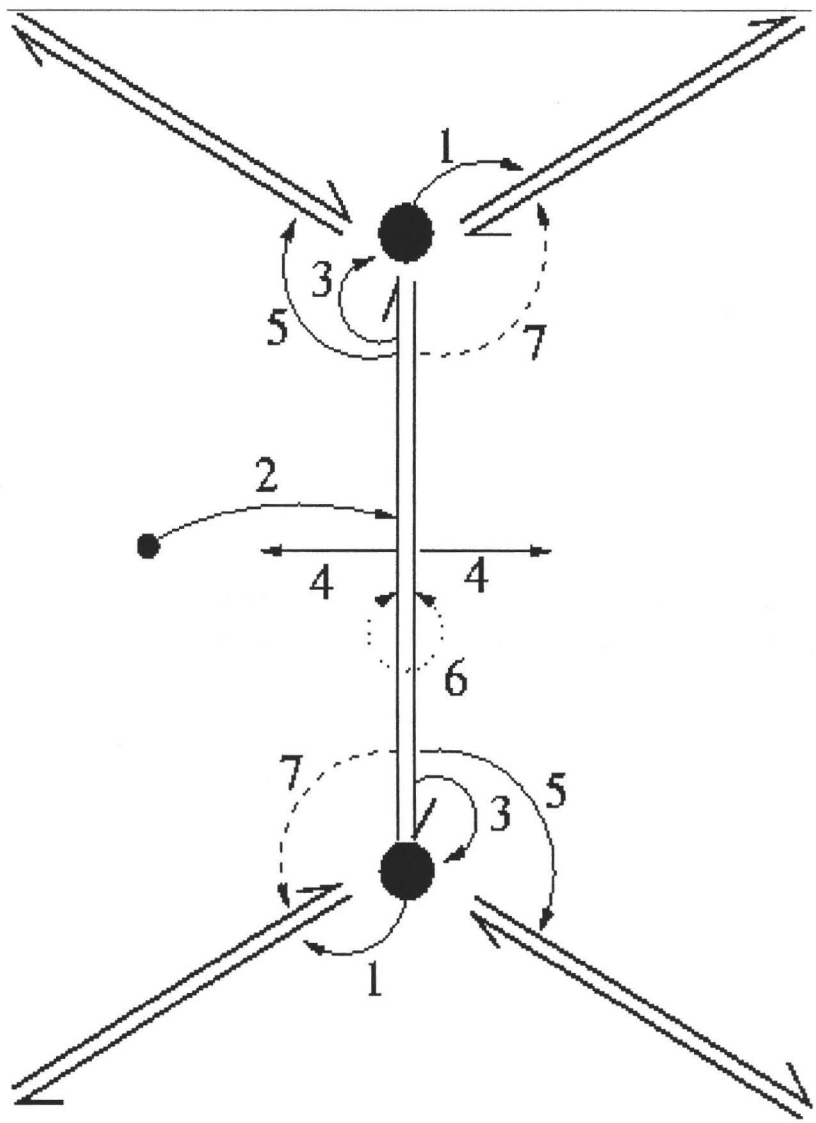

Figure 4.1: Halfedge Data Structure. Image from OpenMesh (Last Access: January, 2006)

with quadrilateral meshes (Cao, 2006). Hence, an edge-based data structure is chosen to improve the *Flexibility* (QF7) of PMGT. The data structure for PMGT is based on the halfedge data structure used by OpenMesh (Last Access: January, 2006). The two differences are as follows:

1. Instead of arbitrary outgoing halfedge stored for a vertex, the first outgoing halfedge is stored. The first outgoing halfedge of a boundary vertex is the outgoing halfedge whose opposite halfedge is a boundary halfedge. A boundary halfedge is a halfedge whose incident face is undefined. The first outgoing halfedge of a non-boundary vertex can be any outgoing halfedge of the vertex.

2. Instead of an arbitrary halfedge stored for a cell, the halfedge with longest distance between the start vertex and the target vertex is stored.

A simple mesh is shown in Figure 4.2. There are 5 vertices ($v1 - v5$), 16 halfedges ($h1 - h16$), and 4 cells ($c1 - c4$). $v1, v2, v3$, and $v4$ are boundary vertices, while $v5$ is not a boundary vertex. $h2, h8, h12$, and $h16$ are boundary edges since the incident cell of these halfedges are undefined (the space that is outside of the input domain). Other halfedges are not boundary edges. The first outgoing halfedge of vertex $v1$ is $h1$ since the opposite halfedge of $h1$ is $h2$, and $h2$ is a boundary halfedge. The first outgoing halfedge of vertex $v5$ can be any one of the halfedges $h5, h4, h10$, or $h14$. By changing the arbitrary halfedge stored for a vertex to the first halfedge, any outgoing halfedges can be found during only one iteration. The longest halfedge of cell $c1$ is $h1$. By changing the arbitrary halfedge stored for a cell to the longest halfedge, looking for the

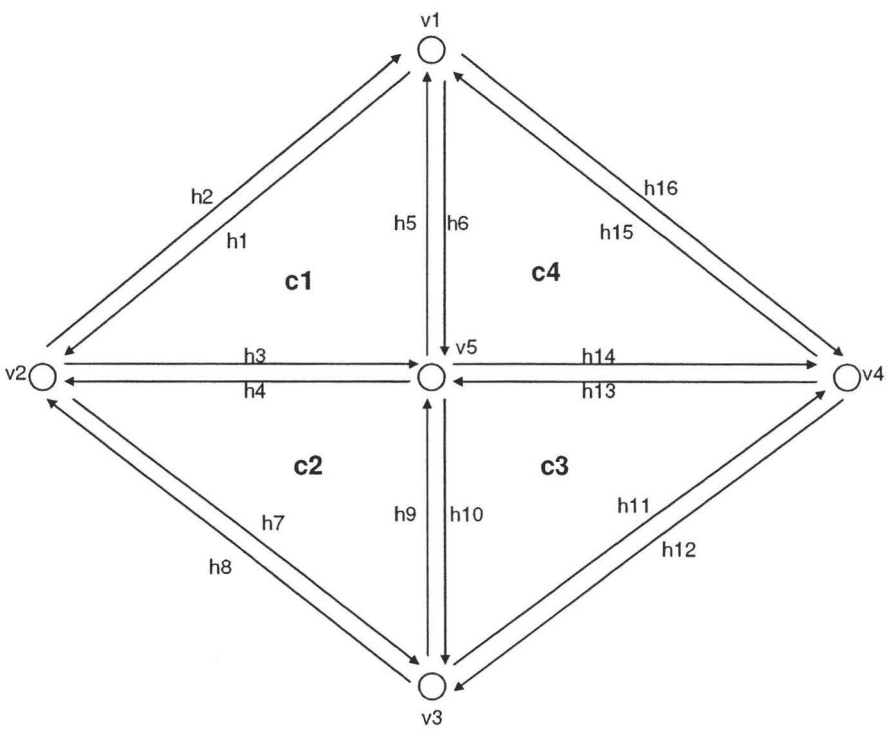

Figure 4.2: Halfedge Data Structure for PMGT.

longest halfedge, which is needed for one of the refining algorithms introduced in Section 4.2, can be done in constant time. Both of above changes improve the *Efficiency* (QF3) of PMGT.

4.2 The Algorithms

According to the SRS, PMGT has two functionalities, refining and coarsening a given mesh. Since PMGT deals with a triangular mesh, only algorithms for triangular meshes are discussed. The algorithms used by PMGT for refining and coarsening a given mesh are given in Section 4.2.1 and Section 4.2.2, respectively.

4.2.1 Refining

The following three principle methods are commonly used for triangle refinement.

- Edge bisection. An example of this method is longest edge (side) bisection. The triangle that is marked for refinement is first bisected by the longest edge into two and if non-conformity (that is, the intersection of any two cells in the mesh is other than one of the following: a vertex, or an edge or empty) still persists, the mesh is further refined to maintain the conformity.

- Point insertion: Usually the point is inserted at the centroid of an existing element. After insertion, the mesh can be refined by dividing the triangle into three triangles.

- Template: One example is to decompose a single triangle into four similar triangles by inserting a new vertex, usually at the midpoint of its edges. If non-conformity exist, then the mesh is further refined to maintain the conformity.

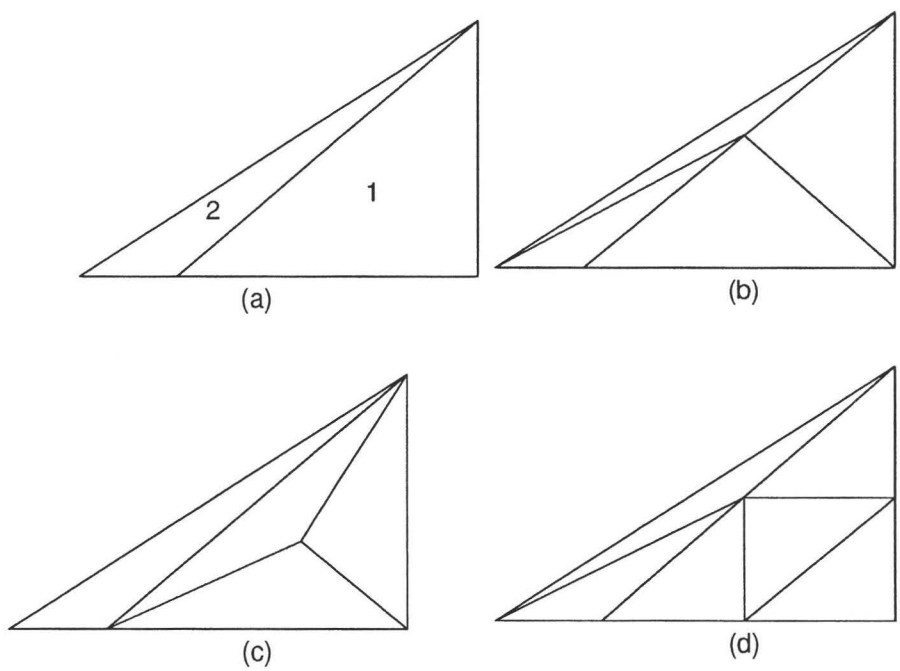

Figure 4.3: An Illustration of the Refining Algorithms

An illustration of above three algorithms is shown in Figure 4.3. A simple input mesh with 2 cells is shown in sub-figure (a). The cell 1 is meshed for refinement while cell 2 is not. The sub-figure (b) is the result of refining (a) using longest edge bisection algorithm. Note that the cell 2 is also refined to maintain conformity. How to refine the cell 2 depends on the algorithm. For simpleness the cell 2 is only divided into two cells. The sub-figure (c) is

the result of refining (a) using point insertion algorithm. There is no non-conformity problem involved; therefore, cell 2 is not refined. The sub-figure (d) is the result of refining (a) using one of the template algorithms, which inserts three points at the midpoints of three edges. The cell 2 is refined to maintain conformity, and the cell 2 is divided into two cells, as for the longest edge bisection algorithm.

All of above algorithms can fulfill the refining requirement provided in the SRS. Due to the limited resource, only the point insertion algorithm and the longest edge bisection algorithm were implemented. The point insertion algorithm used by PMGT is the same as that describing above. However, the longest edge bisection algorithm used by PMGT is different. In the above input mesh, the edge that is the longest edge of cell 1 is not the longest edge of cell 2. Simply dividing cell 2 into two cells, as shown in Figure 4.3, often reduces the minimum angle of the mesh. The minimum angle of a mesh is the smallest value among the smallest angles of the cells in the mesh. Usually, the greater the minimum angle, the better the quality of a mesh. The better the quality of a mesh, the closer the mesh representing the domain. The closer the mesh representing the domain, the more accurate the result of using the mesh to solve a particular problem. Although there is no requirement for the minimal angle of the refined mesh, improving the quality of the refined mesh can improve the *Reliability* (QF2) of PMGT.

Rivara and Inostroza (1995) and Rivara (1997) proposed algorithms to solve the problem of reducing the minimum angle of the mesh. Rivara's algorithms are basically divided into two categories, *pure longest side bisection*

algorithm and *backward longest side bisection algorithm*, respectively. According to Rivara and Inostroza (1995) and Rivara (1997), the pure longest side bisection is outlined as follows, where `T` represents a mesh and `t` represents a cell:

```
Longest-side-bisection (T,t)
Perform a longest-side bisection of t
    (Let P be the point generated)
While P is non-conforming then do
    Find the neighbor t* of t (by the side containing P)
    Longest-side-bisection (T,t*)
```

The above algorithm is recursive. The first action, which is `Perform a longest side bisection of t`, is just dividing `t` into two triangles by adding an edge connecting `P` with the opposite vertex of the longest side. Figure 4.4 illustrates the algorithm. The sub-figure (a) is the initial mesh with cell `t` to be refined. The points 1 and 2 in sub-figure (b) are two intermediate non-conformal generated points. The sub-figure (c) is the final mesh for the algorithm. The sub-figure (d) is the simplified version of the above algorithm, which will be introduced later.

In Rivara (1997), a non-recursive version of the longest side bisection algorithm, called backward longest side bisection algorithm, is proposed as follows:

```
Backward_Longest-Side-Bisection(T,t)
While t remains without being bisected do
```

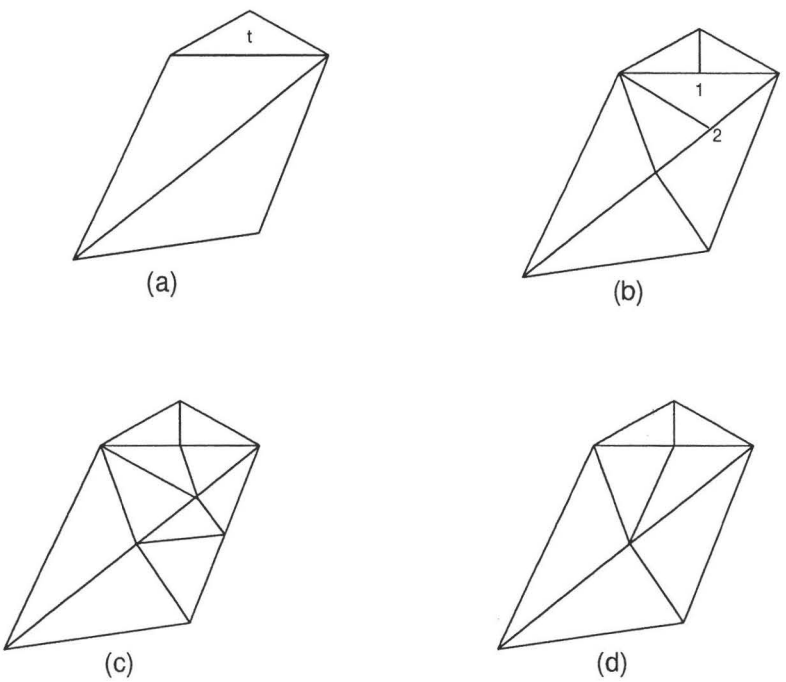

Figure 4.4: An Illustration of the Pure Longest Side Bisection Algorithm Proposed by Rivara and Inostroza (1995)

```
Find the LSPP(t)
    If t*, the last triangle of the LSPP(t), is a
        terminal boundary triangle, bisect t*
    Else bisect the (last) pair of terminal triangles
        of the LSPP(t)
```

The longest side propagation path of a triangle $t(0)$, $LSPP(t(0))$, is defined as the ordered list of all the triangles $\{t(0), t(1), t(2), ..., t(n-1), t(n)\}$, such that $t(i)$ is the neighbor triangle of $t(i-1)$, by the longest-side of $t(i-1)$, for $i = 1, 2, ..., n$. The algorithm is illustration in Figure 4.5. The sub-figure

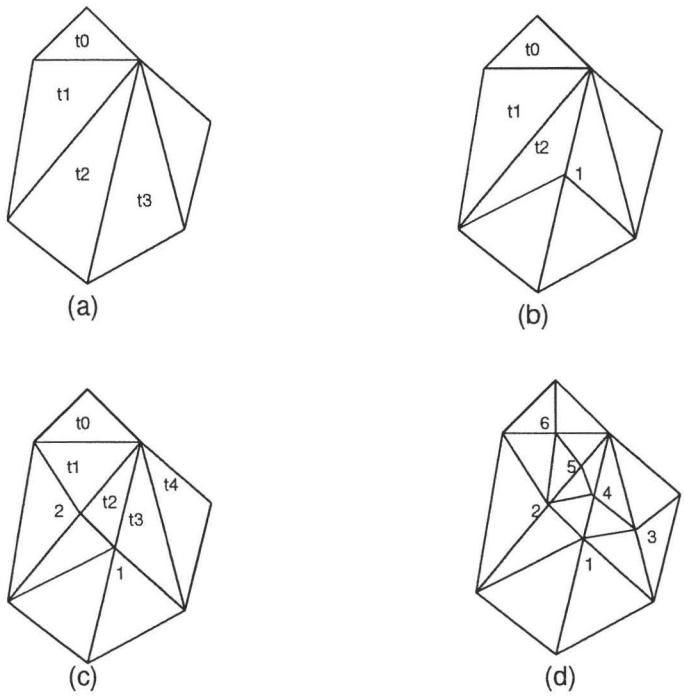

Figure 4.5: An Illustration of the Backward Longest Side Bisection Algorithm Proposed by Rivara (1997)

(a) is the initial mesh with $t0$ to be refined. In Figure 4.5, $LSPP(t0) = \{t0, t1, t2, t3\}$. The $LSPP(t0)$ is not further expanded because the longest edges of $t2$ and $t3$ are the same edge. The sub-figure (b) and (c) illustrate the first 2 steps, and the sub-figure (d) is the final mesh. The new vertices have been enumerated in the order that they were created.

In both the pure longest side bisection algorithm and the backward longest side bisection algorithm, a cell may be refined more than once. However, PMGT does not need to be that complicated. To make the PMGT easier to understand and easier to implement, refining each cell in the mesh

once is enough. In Rivara and Inostroza (1995), a simplified version of the pure longest side bisection algorithm is proposed. The result of this simplified version, which is shown in sub-figure (d) of Figure 4.4, is what PMGT desires. However, it is achieved in Rivara and Inostroza (1995) in a complicated way. By modifying the backward longest side bisection algorithm a new algorithm that creates a simpler resulting mesh can be achieved in an elegant way. The new longest edge bisection algorithm used by PMGT is as follows.

```
procedure Refining(m: mesh)
begin
    for each cell c in m do
        if c is marked to be refined
            find the LSPP(c)
            while LSPP(c) is not empty do
                c' := the last triangle of the LSPP(c)
                mark c' to no-change
                remove c' from LSPP(c);
                if LSPP(c) is empty then
                    bisect c'
                else
                    c'' := the last triangle of the LSPP(c)
                    mark c'' to no-change
                    remove c'' from LSPP(c)
                    if c' is a terminal boundary triangle then
                        bisect(c')
```

```
            c':= a new cell that adjacent to c''
        bisect-pair(c',c'')
        if c'' = c then
            break
        else
            c''' := the last triangle of the LSPP(c)
            cc := a new cell adjacent to c'''
            add cc to the end of the path LSPP(c)
end
```

In the above algorithm, the procedure `bisect-pair(c', c'')` divides adjacent cells `c'` and `c''` into two cells respectively by adding a vertex `v` in the midpoint of the common edge `e` of `c'` and `c''`, and adding edges connecting v and the opposite vertices of `e`. The final mesh after applying this algorithm to the initial mesh shown in the sub-figure (a) of the Figure 4.4 is the mesh shown in the sub-figure (d).

4.2.2 Coarsening

Most mesh coarsening algorithms deal with vertices or edges; that is, either vertice are removed (vertex decimation) (Guillard, 1993; Miller et al., 1997; Ollivier-Gooch, 2003), or edges are removed (edge decimation) (Gueziec, 1995; Ollivier-Gooch, 2003). However, as the SRS shows, PMGT coarsens a given mesh by removing cells. Hence, an existing algorithm needs to be modified to be used by PMGT.

Note that Ollivier-Gooch (2003) is listed in both vertex decimation

and edge decimation. The reason is that it dicusses an algorithm of vertex decimation. However, an edge decimation algorithm is used to remove a vertex. The algorithm is illustrated by Figure 4.6. The left half of the Figure shows a vertex 0, which is to be removed from the mesh, and its immediate neighbors in the mesh. Vertex 0 will be removed by sliding it along the edge $\overline{02}$ to vertex 1. In the process, cells $\Delta 021$ and $\Delta 032$ are removed, as are edges $\overline{01}$, $\overline{02}$ and $\overline{03}$. The resulting mesh fragment is shown in the right half of Figure 4.6.

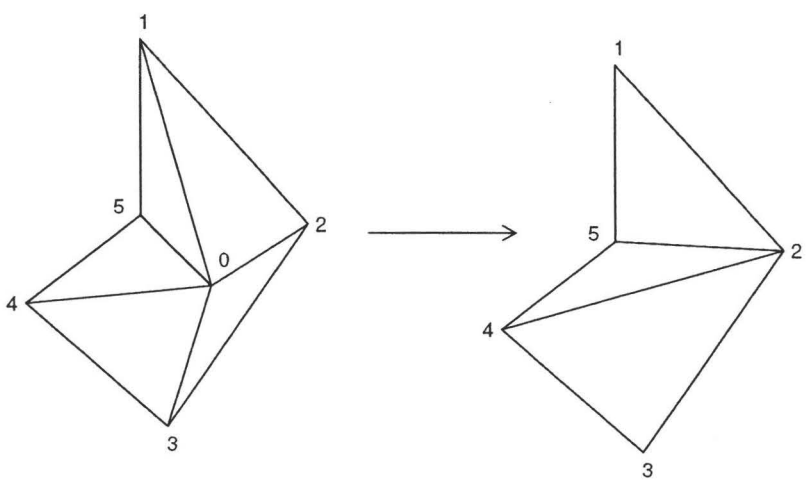

Figure 4.6: An Illustration of the Coarsening Algorithm used by Ollivier-Gooch (2003)

Since the vertex decimation can be done by using an edge decimation algorithm, it is conjectured that cell decimation can also be done by using an

edge decimation algorithm. In fact, the algorithm illustrated above results in two cells being removed. The idea of coarsening algorithm used by PMGT is that a cell, which needs to be removed, and one of its adjacent cell, which also needs to be removed, are removed by removing the edge that is incident to both cells. As far as we know, this idea has not been formally proposed elsewhere. The new algorithm is described below.

```
procedure Coarsen(m: mesh)
begin
    for each cell c in m do
        if c is marked to be removed then
            for each cell c' that is adjacent to c do
                if c' is marked to be removed then
                    for each vertex v that is an end point of the
                    edge e that is incident to both c and c' do
                        v' := another end point of e
                        if legalRemove(v, e) then
                            for each cell cc that is incident
                            to v do
                                mark cc to no-change
                            for each edge eee that is incident
                            to v do
                                if (eee is incident to c
                                or eee is incident to c') then
                                    delete eee;
```

```
                    for each edge ee that is incident
                    to v do
                        update the end point of v to v'
                    delete e
                    delete c, c'
                    delete v
end
```

In above algorithm the function `legalRemove(v, e)` return true if sliding the vertex `v` along the edge `e` to vertex `v'` is legal. This function can be described as follows:

```
function legalRemove(v: vertex, e: edge)
begin
    for each cell c that is incident to v do
        if c is not incident to e then
            for each pair (v1, v2), in which
            v1, v2 are other two vertices
            that are incident to c do
                if v,v1,v2 are in counterclockwise order then
                    v' = another vertex that is incident to e
                if v',v1,v2 are not in counterclockwise order then
                    return false
    return true;
end
```

4.3 The Programming Language

To choose a programming language to implement PMGT, a programming paradigm must be selected first. A programming paradigm is a way of conceptualizing what it means to perform computation and how tasks to be carried out on a computer should be structured and organized (Floyd, 1979). There are four basic programming paradigms, namely imperative programming (Pascal, C), objected oriented programming (C++, Java), functional programming (Haskell, Ocaml), and logic programming (Prolog, Mercury). Functional programming languages do not fit for PMGT, since PMGT is intended for industrial practitioners and functional programming language is more the domain of academics. Logic programming languages also do not seem appropriate for PMGT as the application domain of logic programming language focuses on expert system and automated theorem proving, not on scientific computing applications. Imperative programming languages are a good candidate for PMGT since they are the most widely used programming languages, and because they have the advantage of faster implementations and better tool support (Grabmüller and Hofstedt, 2003). Object oriented (OO) programming languages are also an option because OO can lead to more maintainable programs, since OO programs consist of small self contained parts (classes). In addition, the OO aspect of inheritance enables the programmer to make new versions of a program by only programming the differences between the existing program and the new program.

Although the *Efficiency* (QF3) of PMGT is important, the overall quality of PMGT is more desirable. The OO programming paradigm is selected

to improve the *Flexibility* (QF7), *Maintainability* (QF5) and the *Reusability* (QF9) of PMGT. Between the most widely used OO programming languages, C++ and Java, C++ was chosen. The reason is that in addition to the fact that the C++ is faster than Java in general, C++ is commonly used by mesh generation software developers and by those in scientific computing community. Using C++ to implement PMGT can make the communication with other mesh generation developers easier; therefore, improving the *Usability* (QF4) of PMGT.

4.4 Other Decisions

This section discusses other decisions that relate to the implementation of PMGT. PMGT has the ability to manipulate meshes by taking advantage of parallelism. Section 4.4.1 discusses the decisions made for the parallel version of PMGT. Section 4.4.2 discusses the decisions that relate to the system.

4.4.1 Decisions about Parallelism

PMGT has two versions, a serial version and a parallel version. The use of multiprocessors in the parallel version is to improve the *Efficiency* (QF3) of PMGT since in general, the use of multiprocessors will reduce the execution time.

The data structure for the two versions are similar, except that there is parallel information, such as global id for each cell, given for the parallel version of PMGT. Due to the limited resource of time, only one algorithm is

implemented in the parallel version. The implemented algorithm is the point insertion algorithm for refining a mesh. There is no conformality problem in this algorithm; therefore, no communication is necessary among processors during the refinement. The programming language for implementing the parallel version of PMGT is MPI, since C++ is binded in MPI. The parallel version of PMGT could be written in a complicated language that is design specifically for parallel computation, such Charm++ (Koenig, 2003). However, *Usability* (QF4) of PMGT would not be promoted by this decision.

Before performing refinement the entire initial mesh is loaded into the local memory of each processor and then partitioned according to the number of processors. The reason for this is to simplify the algorithm. The increased loading is not considered to be a problem since the initial mesh is assumed to be simple, thus it will not need much space. The other option would be to first partition the mesh and then only information related to each processor would be stored in that processor. In this case, the space needed to store the mesh would be reduced for each processor. However, the time needed to get its portion for each processor would be increased and more programming time would have to be committed to writing routines to partition vertices, edges, and cells.

4.4.2 Decisions about the System

Some decision relates to the system as specified below to improve the qualities of PMGT.

- For each entity (including vertex, edge, or cell), an id, which is not men-

tioned in the MIS, is stored to improve the *Flexibility* (QF7) of PMGT, since the id can be implemented by different data structures, such as index for array, pointer for linked list. For parallel version, there are two parts of an Id, global id and local id.

- An array (or vector in the language of C++) is used to implement the lists of entities of a mesh in PMGT. To improve the *Efficiency* (QF3) of PMGT, before a entity is deleted, it is switched to the end of the list. Hence, deleting an entity only needs constant time. Adding an entity (to the end of the list) takes constant time, too.

- Real numbers are approximated by IEEE double precision floating-point number. This decision can improve *Reusability* (QF9) of PMGT since this implementation is widely used by most software.

- Use comments in the code, such as header information for all files, and frequently use "ReadMe.txt" to explain the usage of the software. These coding style can improve the *Usability* (QF4) of PMGT.

- The implementation of PMGT is targeted at SHARCNET. However, it can execute on Linux/Unix/Mac operating system with g++ complier for the serial version. For the parallel version, the g++ compiler and MPI library are needed. The serial version could easily be adopted to the Windows OS with minor modification. Implementing PMGT so that it can be used for more environments than just SHARCNET improves the *Portability* (QF8) of PMGT.

4.5 Software Technologies Used to Assist the Implementation

Software engineering technologies were used through the implementation of PMGT to improve the software qualities. These technologies can improve the overall quality of software development process, as well as the software product.

- Version Control. Version control combines procedures and tools to manage different versions of configuration objects that are created during the software engineering process (Pressman, 1999). Version control is done for the entire development of PMGT. Although version control is important in the stage of software requirements, design, and testing, it is most valuable in the implementation, since more modification is involved in this stage. One of the advantages of version control is to easily diagnose errors by comparing incorrect versions of code to correct versions of code. Therefore, version control improves *Maintainability* (QF5). The version control program used by PMGT is subversion.

- Makefile. A makefile can replace several commands, which can be error prone; therefore, the adopting of using a makefile improves the *Correctness* (QF1) of PMGT.

- Namespace. Using namespaces can avoid name conflict and unnecessary access of protected data and routines; therefore, this decision was make to improve the *Correctness* (QF1) of PMGT.

- Traceability Matrix. A traceability matrix of classes and modules can be used to check the completeness and consistency of the implementation against the design. The class-module traceability matrix of PMGT is shown in Figure 4.1. Only the modules that are implemented in PMGT are shown in the matrix. It is can be seen that classes and modules have a one to one relation. This promotes the *Reusability* (QF9) and the *Maintainability* (QF5) of PMGT. The definition of access routines in the vertex module (M8), the edge module (M9), and the cell module (M10) can be found in the file "Entity.h". The definition of the access routines in the input format module (M5), the output format module (M6), and the mesh module (M11) can be found in the file "Mesh.h". The definition of the access routines in the refining module (M12) and the coarsening module (M13) can be found in the file "Algorithms.h". The definition of access routines in the service module (M7) can be found in the file "tester.h".

	M5	M6	M7	M8	M9	M10	M11	M12	M13
Input Class	✓								
Output Class		✓							
Service Class			✓						
Vertex Class				✓					
Edge Class					✓				
Cell Class						✓			
Mesh Class							✓		
Refining Class								✓	
Coarsening Class									✓

Table 4.1: Traceability Matrix: Classes and Modules

Chapter 5

Testing

In general, the purpose of testing is to measure and improve software qualities, which are defined in Section 1.1. However, due to limited resources of time, it is difficult to test PMGT against all of the quality factors. Like other scientific computing software, *Correctness* (QF1) and *Efficiency* (QF3) are among the most important quality factors of PMGT. The testing of PMGT focuses on validating the correctness and efficiency of PMGT. First, the details of what is included in the tests are discussed in Section 5.1. Then, the test cases with respect to the scope of the test are specified in Section 5.2. Finally, the result are recorded and analyzed in Section 5.3. The full details about the validation tests for PMGT are in Appendix D.

5.1 The Scope of the Testing

One of the most challenging aspects of testing the *Correctness* (QF1) of PMGT is that the actual output mesh for a specific input mesh is unknown. The

unknown solution is a challenge that is common to most scientific computing software, as mentioned in Section 1.2. In fact, in other scientific computing software, such as software to solve ordinary or partial differential equations, the challenge is even more pronounced because a unique true solution is being sought. Mesh generation software is different because the notion of a unique true solution does not apply, but it is still necessary to compute a valid solution of the required quality. Moreover, there is still the challenge of the lack of an expected solution for comparison purposes. The lack of an expected solution makes PMGT difficult to test for correctness. Without a known solution, we can still test properties of the calculated solution that we know must be true. One way to test the *Correctness* (QF1) of PMGT is to see whether the output mesh is a refined or coarsened mesh of the input mesh. According to the SRS, the characteristics of a refined mesh relate to the data definition of *Refined* (D23) and that of a coarsened mesh relate to the data definition of *Coarsened* (D24). The data definitions D23 and D24 are defined in the SRS. In both definitions, the output mesh needs to be a valid mesh (D18) and the input mesh and output mesh covers up each other (D19). In addition, other requirements that are common to a mesh, such as that a mesh conforms to the Euler Equation, should also be met.

To improve the *Usability* (QF4) of PMGT, the correctness test is automated. The follows lists the automated correctness validation test requirements (ACVTRs) of PMGT:

- The area of each element is greater than zero (referring to D5).
- The boundary of the mesh is closed. (referring to D15).

- The mesh is conformal (referring to D16).
- The intersection of any two elements is empty (referring to D17).
- The input mesh and output mesh *CoveringUp* each other (referring to D19).
- The length of each edge is greater than zero. (This is required by the definition of a mesh, which is defined in the SRS.)
- The vertices of each element are listed in a counterclockwise order. (The counterclockwise order of the vertices for each element is not necessary for implementing PMGT. However, it is adopted by most meshing and FEA software. PMGT uses this convention.)
- The output mesh conforms to the Euler Equation. (This requirement is not documented in the SRS. However, any mesh should implicitly satisfy the equation $nc + nv - ne = 1$, where nc is the number of cells, nv is the number of vertices, and ne is the number of edges.)

Since the output mesh can be displayed on the screen, the output meshes can also be visually checked to ensure that the following visual correctness validation tests requirements (VCVTRs) are met:

- No vertex is outside of the input domain.
- No vertex is inside of a cell.
- No dangling points or edges are present.

- All cells are connected.
- The mesh is conformal.

Some of the VCVTRs overlap with the ACVTRs. This redundancy provides increased confidence in case one testing method fails to catch an error. Both ACVTRs and VCVTRs improve the *Testability* (QF6) of PMGT.

For the efficiency test, no comparison with other software, such as AOMD, is done, due to the limited resource of time to spend on testing and the difficulty of using other mesh generation software. Only the execution time of PMGT is considered. In particular, the execution of parallel version of refinement with different numbers of processor, and the execution time for the serial version of refinement are measured.

5.2 Test Cases

In the validation test on PMGT, there are five test cases for testing correctness of the serial version. The test cases TC1, TC2, and TC3 refine a given mesh by refining some specific cells and then coarsen it by coarsening the refined cells. The difference among these three test cases is different algorithms or different input meshes. This refining and coarsening process may be performed several times. The test case TC4 refines and coarsens meshes according to the sizes of the cells, which are defined as the length of the shortest edges of the cells. The test cases TC5 repeatedly refines the given mesh by refining the specified cells until the required number of refinements is reached. The test case TC6 tests both the correctness of the parallel version and the efficiency of PMGT.

This test case refines all cells in the mesh several times. To test the execution time, different the numbers of processor are used for the parallel version, and the same algorithm for the parallel version is used for the serial version.

The test cases TC2, TC3, TC4, and TC5 use the longest edge bisection algorithm that is defined in Page 75 for refinement. This longest edge bisection algorithm is call *Refining* for short. The test cases TC1 and TC6 use the point insertion algorithm that is mentioned in Section 4.2.1 for refinement. This point insertion algorithm is called *Spliting* for short. The coarsening algorithm used is defined on Page 78.

5.3 Results and Analysis

The details of the test results are given in Appendix D. To illustrate the test process the results for two selected test cases are produced here. The analysis of the the result is also included here. The analysis discusses the speedups of different numbers of processors and the traceability matrices.

5.3.1 Selected Results

Two test cases are selected to illustrate the test results. One is the test case TC1. The input mesh is showed in Figure 5.1. The refining and coarsening criterion is that the cells that intersect with the vertical line, $x = 0.6$, are *Split* once, then the cells of the new mesh that intersect with the vertical line are coarsened once. When the splitting and coarsening is done, the vertical line is moved to the right one unit ($x = x + 1.0$), and another Splitting and

Test Case Number	TC1
Test Case Name	SplitCS
Input	Figure 5.1
Expected Output	ACVTRs and VCVTRs listed in Section 5.1 are met
Actual Output	Summary of the correctness test: 15 tests are performed. 15 tests succeed. 0 tests fail.
Selected Output Mesh	Figure 5.2, 5.3, 5.4
Result	Passed

Table 5.1: Test Case1

coarsening is performed. This procedure is until no cells intersect with the vertical line. The test cast TC1 is illustrated in Table 5.1.

The other selected test case is TC6, which is shown in Table 5.2. This test case tests both the correctness and speed of PMGT. The input mesh is shown in Figure 5.5. This test simply splits all cells of the mesh 4 times. It is done in both the serial version and the parallel version with different numbers of processors. The execution time of setting the cells to be refined and splitting the cells is measured. The time spent on input and output is not included.

5.3.2 Analysis

All of the test cases conform to the ACVTRs and VCVTRs listed in Section 5.1. The test result of TC6 show that when the number of cells increased, the execution time increased, and when the number of processors increased, the execution time decreased. That is, this test is passed. Figure 5.7 show the speedup when using different numbers of processors. The speedup is defined

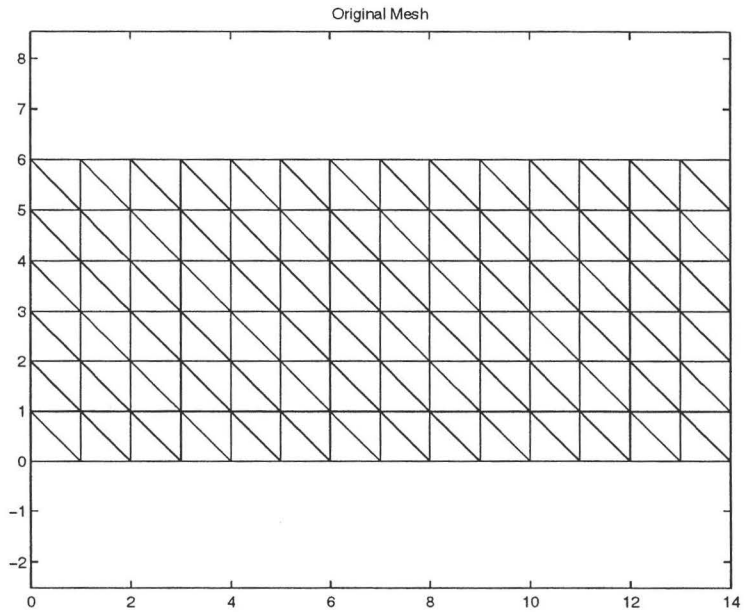

Figure 5.1: Input 1

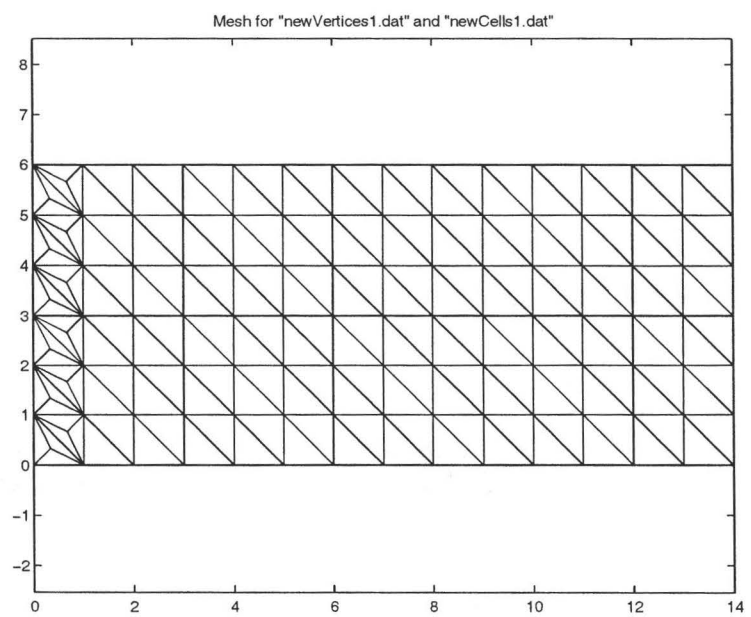

Figure 5.2: Output 1 of TC1

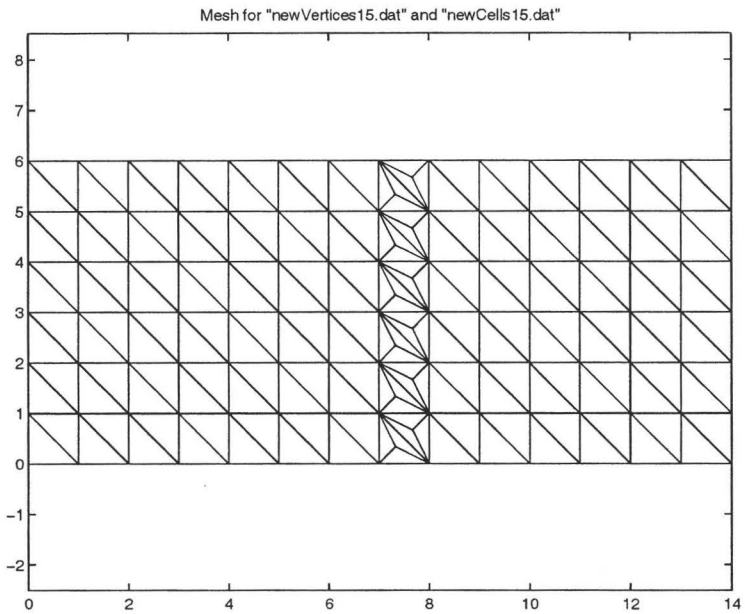

Figure 5.3: Output 2 of TC1

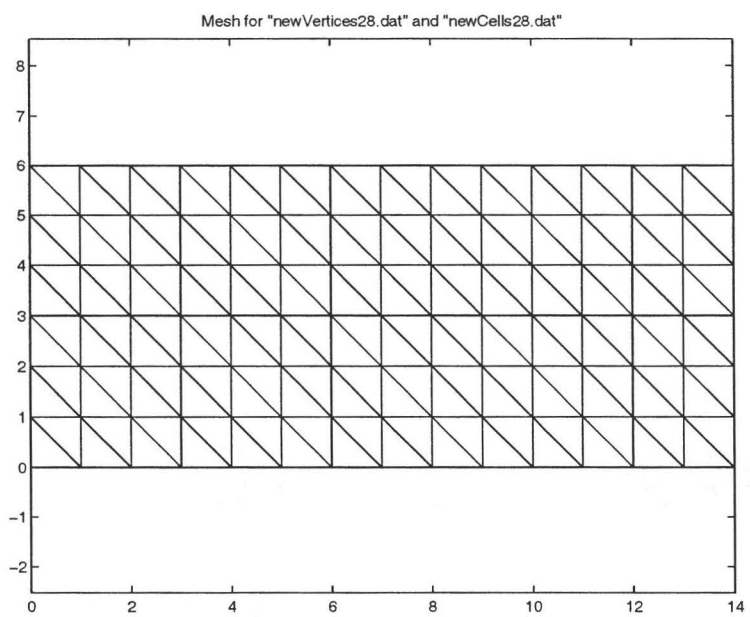

Figure 5.4: Output 3 of TC1

Test Case Number	TC6
Test Case Name	SplitM
Input	Figure 5.5
Expected Output	ACVTRs and VCVTRs listed in Section 5.1 are met Execution time increases as the number of cells increases. Execution time decreases as the number of processors increases.
Actual Output	Execution time as indicated in Figure 5.6
Selected Output Mesh	The mesh is too dense to be shown.
Result	Passed

Table 5.2: Test Case 6

as

$$Speedup(n) = \frac{T_1}{T_n}$$

Where T_1 is the execution time of the serial version, and T_n is the execution time of the parallel version with n processors. In general, $Speedup(n) < n$. However, for PMGT, when the number of cells is greater than 2700, $Speedup(n) > n$, which represents a super linear speedup. Since the algorithms used for the serial version and the parallel version are the same, the super linear speedup is probably due to the cache effect. That is, when the numbers of processors increases, the size of the accumulated caches from different processors also increases. With the larger accumulated cache size, more, or even all, core data set can fit into the caches and the memory access time reduces dramatically. This may explain the extra speedup in addition to the speedup from parallelization of the computation.

In the traceability matrix for software requirements, if a test case tests

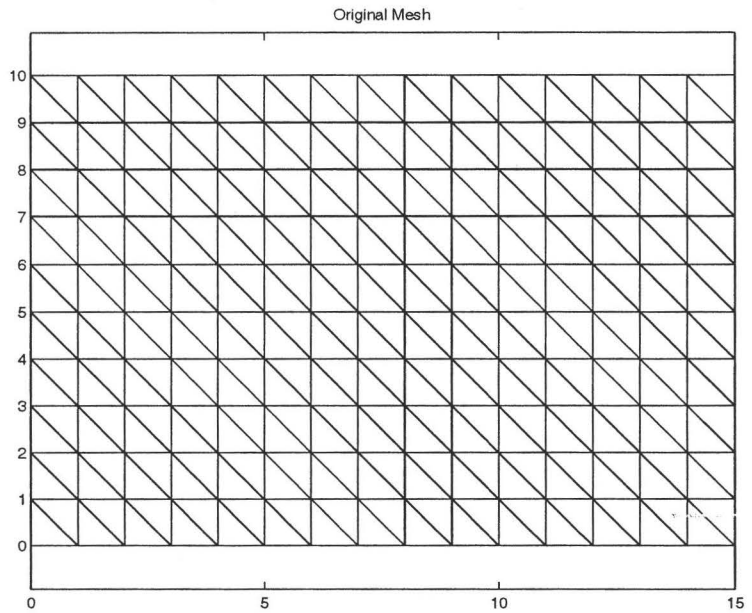

Figure 5.5: Input of TC6

the functionality of a software requirement, there will be a check mark on the cell for the corresponding test case and software requirement. In each row of the traceability matrix for software requirements (Table 5.3), if the requirement in that row defines the correctness or the speed of the software, one or more cells in this row are checked. Otherwise, all cells in the row are empty. Table 5.3 shows that the test cases developed assist with validating the correctness and speed of the software.

Similar to Table 5.3, the traceability matrix for the modules (Table 5.4) shows that the test cases validate the modules that are associated with correctness and speed.

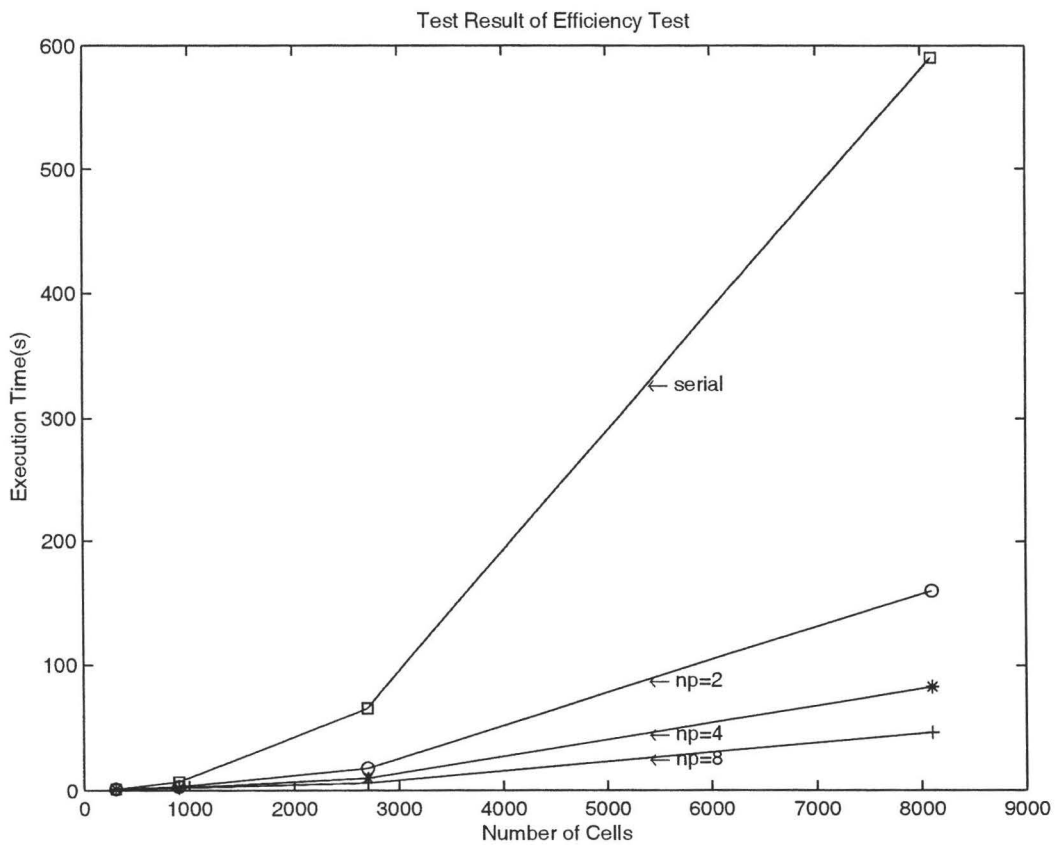

Figure 5.6: Output of TC6

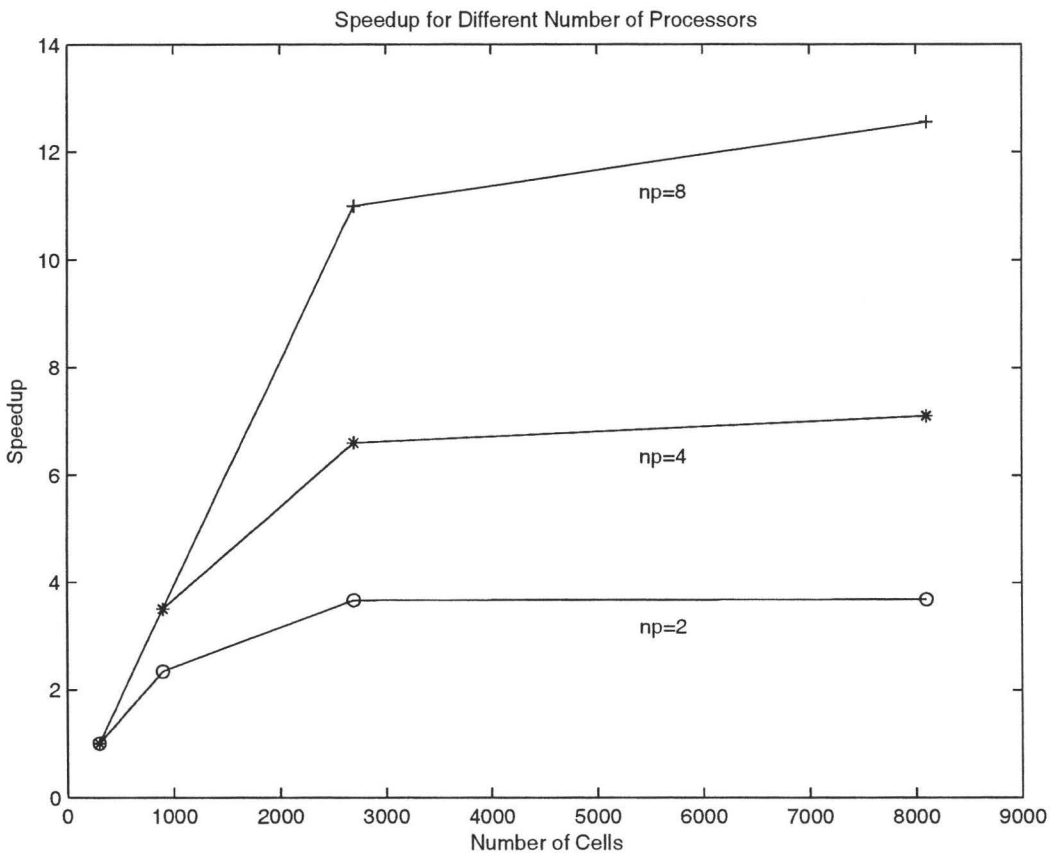

Figure 5.7: Speedup for Different Numbers of Processors

	TC1	TC2	TC3	TC4	TC5	TC6
F1	✓	✓	✓	✓	✓	✓
F2	✓	✓	✓	✓		
F3	✓	✓	✓	✓	✓	✓
F4	✓	✓	✓	✓	✓	
F5	✓	✓	✓	✓	✓	
F6	✓	✓	✓	✓	✓	
F7	✓	✓	✓	✓	✓	
F8	✓	✓	✓	✓	✓	✓
F9	✓	✓	✓	✓	✓	✓
F10	✓	✓	✓	✓	✓	
F11	✓	✓	✓	✓	✓	
F12	✓	✓	✓	✓	✓	
F13	✓	✓	✓	✓	✓	
F14	✓	✓	✓	✓	✓	
F15	✓	✓	✓	✓	✓	
F16						
N1						✓
N2						
N3						
N4						
N5						
N6						
N7						

Table 5.3: Traceability Matrix: Test Cases and Requirements

	TC1	TC2	TC3	TC4	TC5	TC6
M1	✓	✓	✓	✓	✓	✓
M2	✓	✓	✓	✓	✓	✓
M3	✓	✓	✓	✓	✓	✓
M4	✓	✓	✓	✓	✓	✓
M5	✓	✓	✓	✓	✓	✓
M6	✓	✓	✓	✓	✓	✓
M7	✓	✓	✓	✓	✓	
M8	✓	✓	✓	✓	✓	✓
M9	✓	✓	✓	✓	✓	✓
M10	✓	✓	✓	✓	✓	✓
M11	✓	✓	✓	✓	✓	✓
M12	✓	✓	✓	✓	✓	✓
M13	✓	✓	✓	✓		

Table 5.4: Traceability Matrix: Test Cases and Modules

Chapter 6

Conclusions and Future Work

Correctness and efficiency are important for scientific computing software. Other software quality factors, such as those discussed in Chapter 1, also contribute to the software quality, but they are often neglected by developers of scientific computing software. Software engineering methodologies can improve the overall quality of software by considering all software quality factors during the software development process. However, scientific computing software is usually developed by domain experts, who often lack knowledge of software engineering methodologies; therefore, these methodologies are seldom adopted by scientific computing software development community. As a result, there is still room for improvement in the quality of scientific computing software. This thesis attempts to provide an example of developing quality scientific computing software, PMGT, using software engineering methodologies. Hopefully, our work can attract domain experts' interests in developing quality scientific computing software using software engineering methodolo-

gies, such as document driven development. Moreover, the challenges posed for developing scientific software may attract some software engineers to turn their attention toward research on adapting existing methodologies for scientific computing software applications.

In this final chapter of the thesis, the contributions are first summarized in Section 6.1. This is followed by suggested future work in Section 6.2.

6.1 Contributions

In addition to the toolbox itself, our work illustrates that a document driven methodology can improve the overall quality of scientific computing software. This section provides a summary of the relevant software qualities and how the proposed methodology was used to improve them. These qualities relate to the quality factors listed in Section 1.2.

1. Correctness (QF1): The correctness of PMGT is improved by
    - the requirements document (SRS);
    - the use of a *makefile*, which can reduce errors in the implementation;
    - the use of a *namespace*, which can avoid naming conflicts in the implementation;
    - the correctness validation testing;
    - the traceability matrices, since these matrices can be used to ensure that decisions propagate properly through the project.
2. Reliability (QF2): The reliability of PMGT is improved by

- the use of the longest edge refining algorithm, which can improve the minimum angle of the mesh.

3. Efficiency (QF3)

   - the use of the halfedge data structure, in which the outgoing halfedge of a vertex is the first outgoing halfedge and the halfedge of a cell is the longest halfedge;
   - the use of arrays to implement the list of mesh entities;
   - the use of multiprocessors in the parallel version of PMGT.

4. Usability (QF4): All of the documents can improve the usability of PMGT since these documents explicitly tell the reader what PMGT can do at different level of abstraction. In particular, the documents improve the usability of PMGT by:

   - assigning names and unique numbers for items, such as goals, data definitions, software requirements, and modules;
   - decomposition of the system into modules since the decomposition can make PMGT easier to understand;
   - the English explanations for the mathematical notations in the semantics of the access routines;
   - the selection of C++ programming language, which is widely used by mesh generation software developers;
   - the use of the popular MPI library to implement the parallel version of PMGT;

- style adopted for coding, including header information for all files and frequent use of ReadMe.txt files;
- the automation of the correctness validation tests.

5. Maintainability (QF5): In addition to the use of traceability matrices, which facilitate locating errors, the maintainability of PMGT is improved by

   - the SRS, which one can use for finding possible errors;
   - the decomposition of the system into simple and independent modules;
   - the design of the data structure, which is easy to understand;
   - the selection of an OO programming languages, which encapsulates and facilitates finding errors;
   - the use of version control, which keeps the history of the development;

6. Testability (QF6): Unambiguous and validatable software requirements documented in the SRS is the key issue for testability of PMGT. In particular, the testability of PMGT is improved by:

   - the quantifying of the software requirements;
   - the use of formal mathematics in the SRS;
   - the decomposition of the system into simple and independent modules;

- the formality of the semantics of the access routines in the MIS;
- the use of the Automated Correctness Validation Testing Requirements (ACVTRs).

7. Flexibility (QF7): The traceability matrices improve the flexibility of PMGT, since when the software needs to be changed, it is can be done by tracing from the goal down to the file which contains the appropriate module. In addition, the flexibility of PMGT is improved by

   - the waiting rooms in the SRS, which lists the most likely changes of PMGT;
   - the decomposition of the system into simple and independent module;
   - the lists of anticipate and unlikely changes, which can help to find modules that are likely to change, or explicitly identify those maintenance tasks that would not be likely nor feasible;
   - the design of the data structure, which is easy to understand and extend to accommodate a mixed mesh;
   - the selection of an OO programming languages, which makes it easy to change and to add new members;
   - the use of *id* field for each mesh entities within the implementation.

8. Portability (QF8): The portability of PMGT is improved by

   - the system constraints that are specified in the SRS, which states to not focus on details of SHARCNET;

- the decision made during the implementation that PMGT can be executed on Linux/Unix/Mac operating systems.

9. Reusability (QF9) The Reusability of PMGT is improved by

   - the SRS since what the software does can be easily obtained from the SRS;
   - the decomposition of the system into simple and independent modules;
   - the definition of module as work assignments;
   - the selection of OO programming languages, which allows encapsulation;
   - the implementation of the approximation of real numbers as IEEE double precision floating point numbers, which is a choice widely adopted by software developers.

6.2 Future Work

The results of our work encourage further research in the field of using software engineering methodologies to improve quality of scientific computing software. The suggested investigations needed to evaluate the effectiveness of our work are as follows:

- Add more algorithms to enable PMGT to generate meshes with different requirements. One example of these requirements is generating meshes whose minimum angle is greater than a given angle.

- Embed PMGT in a real application, such as a finite element application.
- Test PMGT against more requirements and quality factors.
- Add unit testing for PMGT.
- Compare PMGT to other mesh generation software.
- Modify the service module to allow automated testing as desired by the user, even at run time.
- Add a mechanism dealing with load balancing when communication is needed between processors to refine/coarsen a mesh in parallel.
- Develop a tool to help maintain consistency between the documents

Bibliography

Brian Bauer. Documenting complicated programs. Technical Report CRL Report 316, Department of Computing and Software, McMaster University, 1995.

Barry W. Beohm. A spiral model for software development and enhancement. In *Computer*, volume vol. 21, no. 5, pages pp.61 – 72, May 1988.

Guntram Berti. Generic components for grid data structures and algorithms with C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

Blackpawn. Point in triangle test, Last Access: January, 2006. URL `http://www.blackpawn.com/texts/pointinpoly/default.html`.

Fang Cao. A program family approach to developing mesh generators. Master's thesis, McMaster University, April 2006.

Chien-Hsien Chen. A software engineering approach to developing mesh generators. Master's thesis, McMaster University, Novermber 2003.

Joseph Cirincione. The performance of the patriot missile in the gulf war, October 1992.

Lee Copeland. *A Practioner's Guide to Software Test Design.* Artech House Publisher, 2003.

CSTE. *2006 Guide to the CSTE COMMON BODY OF KNOWLEDGE.* Quality Assurance Institute, 2006.

Alan M. Davis. *Software Refquirements: Analysis and Specification.* Prentice Hall Inc., 1990.

E. W. Dijkstra. *Structured Programming, Chapter Notes on Structured Programming.* Academic Press, London, 1972.

A. H. ElSheikh, S. Smith, and S. E. Chidiac. Semi-formal design of reliable mesh generation systems. *Adv. Eng. Softw.*, 35(12):827–841, 2004. ISSN 0965-9978.

Robert W. Floyd. The paradigms of programming. *Commun. ACM*, 22(8): 455–460, 1979. ISSN 0001-0782.

W. Randolph Franklin. Pnpoly - point inclusion in polygon test, Last Access: January, 2006. URL `http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html`.

Pascal Jean Frey and Paul-Louis George. *Mesh generation Application to Finite Elements.* Hermes Science Europe ltd., 2000.

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Funcamentals of software Engineering.* Pearson Education, Inc., Upper Saddle River, New Jersey 07458, 2003.

Martin Grabmüller and Petra Hofstedt. Turtle: A Constraint Imperative Programming Language. In Frans Coenen, Alun Preece, and Ann Macintosh, editors, *Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, number XX in Research and Development in Intelligent Systems, Cambridge, UK, December 2003. British Computer Society, Springer-Verlag. ISBN 1-85233-780-X.

David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag New Yourk, Inc., 1993.

Andre Gueziec. Surface simplification with variable tolerance. In *Second Annual Intl. Symp. on Medical Robotics and Computer Assisted Surgery (MRCAS '95)*, pages 132–139, November 1995.

H. Guillard. Node-nested multi-grid with delaunay coarsening, 1993.

Daniel Hoffman and Paul Strooper. *Software Design, Automated Testing and Maintenance.* International Thomson Computer Press, 1999.

IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Computer Society, Washington, DC, USA, 1990.

IEEE. *IEEE Guide for Developing System Requirements Specifications.* IEEE Computer Society, Washington, DC, USA, 1998.

IEEE. *IEEE Recommended Practice for Software Requirements Specifications.* IEEE Computer Society, Washington, DC, USA, 2nd edition, 2000.

Gregory Allen Koenig. An efficient implementation of Charm++ on Virtual Machine Interface. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

Lei Lai. Requirements documentation for engineering mechanics software: Guidelines, template and a case study. Master's thesis, McMaster University, Sept. 2004.

William E Lewis and Cunasakaran Veerapollai. *Software testing and continuous quality improvement 2nd ed.* CRC Press LLC, 2004.

J. McCall, P. Richards, and G. Walters. *Factors in Software Quality.* NTIS AD-A049-014, 015, 055, Novermber 1997.

Miller, Talmor, and Teng. Optimal good-aspect-ratio coarsening for unstructured meshes. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

Carl F. Ollivier-Gooch. Coarsening unstructured mesh by edge constraction. *International Journal for Numerical Methods in Edgineering*, 57(3):391–414, May 2003.

OpenMesh. Openmesh, Last Access: January, 2006. URL `http://www.openmesh.org/`.

Steven J. Owen. A survey of unstructured mesh generation technology. In *Proceedings 7th International Meshing Roundtable*, Dearborn, MI, October 1998. doi: www.andrew.cmu.edu/user/sowen/survey.

Steven J. Owen. Meshing research corner, Last Access: January, 2006. URL `http://www.andrew.cmu.edu/user/sowen/mesh.html`.

D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 408–417, Piscataway, NJ, USA, 1984. IEEE Press. ISBN 0-8186-0528-6.

David L. Parnas. On the criteria to be used in decomposing system into modules. *Communications of th ACM*, vol. 15, No. 12:pp.1053 - 1058, December 1972.

David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 1976.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

Roger S. Pressman. *Software Engineering, A Practitioner's Approach, fourth Edition.* McGraw-Hill, 1999.

L.B.S. Raccoon. The chaos model and the chaos life cycle. In *ACM Software Engineering Notes*, volume vol. 20, no. 1, pages pp.55 – 66, January 1995.

M. Rivara and P. Inostroza. A discussion on mixed (longest side midpoint insertion) delaunay techniques for the triangulation refinement problem, 1995.

Maria-Celilia Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulation. *International Journal for Numerical Methods in Engineering*, 40:3313–3324, 1997.

James Robertson and Suzanne Robertson. Volere requirements specification template, 2001.

W.W. Royce. Managing the development of large software system: Concepts and techniques. In *Proc. ICSE.* Computer Society Press, August 1970.

Jim Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 83–92, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics. ISBN 0-89871-313-7.

SCOREC. Algorithm oriented mesh database, Last Access: January, 2006. URL `http://www.scorec.rpi.edu/AOMD/`.

SHARCNET. Shared hierarchical academic research computing network, Last Access: January, 2006. URL `www.sharcnet.ca`.

Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, 1995.

Jonathan Shewchuk. Triangle, a two-dimensional quality mesh generator and delaunay triangulator, Last Access: January, 2006. URL `http://www.cs.cmu.edu/~quake/triangle.html`.

S. Smith and C. H. Chen. Commonality analysis for mesh generation system. Technical Report CAS-04-10-ss, Department of Computing and Software, McMaster University, 2004.

W. Spencer Smith and Lei Lai. A new requirements template for scientific computing. In J. Ralyté, P. Ågerfalk, and N. Kraiem, editors, *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05*, pages 107–121, Paris, France, 2005. In conjunction with 13th IEEE International Requirements Engineering Conference.

W. Spencer Smith, Lei Lai, and Ridha Khedri. Requirements analysis for engineering computation: A systematic approach for improving software reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation*, 13:83–107, 2007.

Ian Sommerville. *Software Engineering.* Addison-Wesley Publishing Company, 1992.

Ian Sommerville and Pete Sawyer. *Requirements Engineering A Good Practice Guide.* John Wiley and Sons, 1997.

Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the fifth IEEE International Symposium on Requirements Engineering*, pages 249–263. IEEE Computer Society, Washington, DC, USA, 2001.

Gregory V. Wilson. Where's the real bottleneck in scientific computing: Scientists would do well to pick up some tools widely used in the software industry. *American Scientist*, 94(1), January – Febrary 2006.

O. C. Zienkiewicz, R. L Taylor, and J. Z. Zhu. *The Finite Element Method Its Basis and Fundamentals.* Elsevier Butterworth-Heinemann, 6th edition, 2005.

Appendix A

Software Requirements Specification for a Parallel Mesh Generation Toolbox

A.1 Reference Material

A.1.1 Table of Symbols, Abbreviations and Acronyms

A.1.1.1 Symbols

Ω	a closed bounded domain in $\mathbb{R}^2$
Ω^*	a mesh covering the domain bounded by Ω
K	a simple shape, such as a line segment in 1D, a triangle or a quadrilateral in 2D, or a tetrahedron or hexahedron in 3D
M^{IN}	an input mesh
M^{OUT}	an output mesh
I	instructions on how a mesh should be refined/coarsened

A.1.1.2 Abbreviations and Acronyms

1D	One Dimensional Space
2D	Two Dimensional Space
3D	Three Dimensional Space
FEA	Finite Element Analysis
HPC	High Performance Computing
PDE	Partial Differential Equation
PMGT	Parallel Mesh Generation Toolbox
SHARCNET	Shared Hierarchical Academic Research Computing Network
SRS	Software Requirements Specification
AOMD	Algorithm Oriented Mesh Database

A.1.2 Index of Requirements

A.2 Introduction

This section gives an overview of the Software Requirements Specification (SRS) for a Parallel Mesh Generation Toolbox (PMGT). First, the purpose of the document is provided. Second, the scope of PMGT is identified. Third, some terminology for software engineering and mesh generation are defined. Finally, the organization of the document is summarized. The Table of Symbols, Abbreviation and Acronyms, and Index of Requirement are given at the beginning of the SRS.

A.2.1 Purpose of the Document

This SRS provides a black-box description of PMGT. The intended audience of the SRS is the development team and the users of PMGT.

A.2.2 Scope of the Software Product

PMGT provides a library that will be embedded into a larger application, such as a finite element analysis (FEA) program.

- The input of PMGT is an existing mesh M^{IN} with instructions I provided by the user on how the mesh should be refined/coarsened.
- PMGT refines/coarsens M^{IN} according to the supplied instructions I on how the mesh should be refined/coarsened.
- PMGT will take advantage of parallel computation.
- The output of PMGT is a refined/coarsened mesh M^{OUT}.

Note that depending on the given instruction, PMGT can either refine or coarsen the given mesh, but cannot do both at the same time. That is, any individual transition from M^{IN} to M^{OUT} will only do one of refining or coarsening. The embedding application will have access to reading the mesh information, such as information on the position of vertices and on the vertices that define a given element. However, the application cannot directly change any mesh data, except for the information indicating which elements should be refined/coarsened.

A.2.3 Terminology Definition

This subsection provides the definitions for terminology used in the SRS. There are two classes of terminology. One relates to software engineering, and the other relates to mesh generation. The definitions are listed in alphabetical order.

A.2.3.1 Software Engineering Related Terminology

Constraint: A statement that expresses measurable bounds for an element or function of the system. That is, a constraint is a factor that is imposed on the solution by force or compulsion and may limit or modify the design changes. (IEEE, 1998)

Context: The boundaries between the system that we intend to build and the people, organizations, other system and pieces of technology that have a direct interface with the system. (Robertson and Robertson, 2001)

Functional Requirements: Functional requirements define precisely what input are expected by the software, what outputs will be generated by the software, and the details of relationships that exist between those inputs and outputs. In short, functional requirements describe all aspects of interface between the software and its environment (that is, hardware, humans, and other software). (Davis, 1990)

Goal: Goals capture, at different levels of abstraction, the various objectives the system under consideration should achieve. (van Lamsweerde, 2001)

Non-functional Requirements: Non-functional requirements define the overall qualities or attributes to be exhibited by the resulting software system. (Davis, 1990)

Requirements: A software requirement is: *i*) a condition or capability needed by a user to solve a problem or achieve an objective; *ii*) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document; or, *iii*) a documented representation of a condition or capability as in the above two definitions. (IEEE, 2000)

Software Engineering: Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. (IEEE, 1990)

Software Requirements Specification: A Software Requirements Specification (SRS) is a document containing a complete description of what the software will do without describing how it will do it. (Davis, 1990)

System: An interdependent group of people, objects, and procedures constituted to achieve defined objectives or some operational role by performing specified functions. (IEEE, 1998)

System Context: System Context documents the relationships between the system being specified and other human and computer systems. (Sommerville, 1992)

User: The person, or persons, who operate or interact directly with the product. (IEEE, 2000)

A.2.3.2 Mesh Generation Related Terminology

Cell: Another name for an element, as defined in page 120.

Conformal Mesh: A conformal mesh is a mesh (defined on page 121) following the definition of a mesh, with the addition of the following property: The intersection of two elements in the mesh Ω^* is either the empty set, a vertex, an edge or a face (when the dimension is 3). (Frey and George, 2000)

Connectivity: There are two types of connectivity, one for the mesh and one for a mesh element:

1. "The connectivity of a mesh is the definition of the connection between its vertices." (Frey and George, 2000)
2. "The connectivity of a mesh element is the definition of the connections between the vertices at the element level." (Frey and George, 2000)

Domain: The area or volume that is to be discretized. The domain is sometimes referred to as the computational domain. (Smith and Chen, 2004)

Edge: An edge is a line segment between two vertices.

Element : The original domain is discretized into smaller, usually simpler, shapes called elements. The typical shapes for elements in 1D is a line, in 2D is a triangle or a quadrilateral, and in 3D a tetrahedron or a hexahedron. Elements are also called cells. (Smith and Chen, 2004)

Embedding Application: The software that uses PMGT.

Face: A face is a maximal connected subset of the plane without vertices inside the subset. In 2D, a face is a cell.(Frey and George, 2000)

Hybrid Mesh: A mesh is said to be hybrid if it includes some elements with a different spatial dimension. (Frey and George, 2000)

Mesh: In Smith and Chen (2004), a mesh is defined as follows:
Let Ω be a closed bounded domain in $\mathbb{R}$ or $\mathbb{R}^2$ or $\mathbb{R}^3$ and let K be an element. A mesh of Ω, denoted by Ω^*, has the following properties:

1. $\Omega \approx \cup(K|K \in \Omega^* : K)$, where $\cup$ is first closed and then opened
2. the length of every element K, of dimension 1, in Ω^* is greater than zero
3. the interior of every element K, of dimension 2 or greater, in Ω^* is nonempty
4. the intersection of the interior of two elements is empty

The only difference between above definition and the definition given by Frey and George (2000) is that equality ($=$) had been changed to approximate equality ($\approx$).

Mesh Generation: The automatic mesh generation problem is that of attempting to define a set of elements to best describe a geometric domain, subject to various element size and shape criteria. (Smith and Chen, 2004)

Mixed Mesh: A mesh is said to be mixed if it includes some elements of a different geometric nature. (Frey and George, 2000)

Structured Mesh: The mesh in which the local organization of the grid points and the form of the grid cells do not depend on their position but are defined by a general rule. There is a pattern to the topology that repeats. Frey and George (2000) say, "a mesh is called structured if its connectivity is of the finite difference type." They go on to remark, "Peculiar meshes other than quad or hex meshes could have a structured connectivity. For instance, one can consider a classical grid of quads where each of them are subdivided into two triangles using the same subdivision pattern."

Topology: "The topology of a mesh element is the definition of this element in terms of its faces and edges, these last two being defined in terms of the element's vertices." (Frey and George, 2000)
The topology of a mesh is the set of topologies of its constitute mesh elements.

Unstructured Mesh: The mesh whose element connectivity of the neighbouring grid vertices varies from point to point. Any mesh that is not structured is an unstructured mesh. (Smith and Chen, 2004)

Vertices: The locations that define the shape of the cells. In 1D the vertices are the end-points of the elements. For 2D and 3D elements the vertices correspond to the location in space that defines the intersection of the edges of an element. (Smith and Chen, 2004)

A.2.4 Organization of the Document

This SRS follows the template introduced by Lai (2004). Lai's template targets an SRS for scientific computing software. In particular, the example shown is for engineering mechanics software, such as software to analyze beams. In the current work, Lai's template is modified to fit PMGT, which is a more general purpose software. For example, the instanced model section of Lai's template is removed since PMGT is not designed for solving a specific physical problem.

Section A.2 (this section) is an introduction to the SRS. The rest of the document is arranged as follows. Section A.3 provides the general information about the system. Section A.4 is the major part of the SRS. All functional requirements and non-functional requirements of the software are presented in this section. Section A.5 discusses some other system issues. Section A.6 gives a traceability matrix that summaries the association of each requirement with goals, assumptions, theoretical models and data definitions introduced in A.4. This SRS also contains the list of possible changes in the requirements and values of auxiliary constants. The references are listed at the end of this document.

A.3 General System Description

This section describes the general information about the system. The interfaces between the system and its environment are defined first. Then the characteristics of potential users are discussed. At end of this section, some system constraints are described.

A.3.1 System Context

The software to be built is a library tool that will be called by other applications. There is no direct interaction between the system and the end users. Users of the embedding application, such as an FEA program, provide some parameters directly to the FEA program. Some of these parameters are passed to PMGT by the FEA program. The interface between PMGT and the embedding application should only show what PMGT can do and hide the information about how to do it. Therefore, users who are not experts in mesh generation or in parallel processing will be able to use this toolbox.

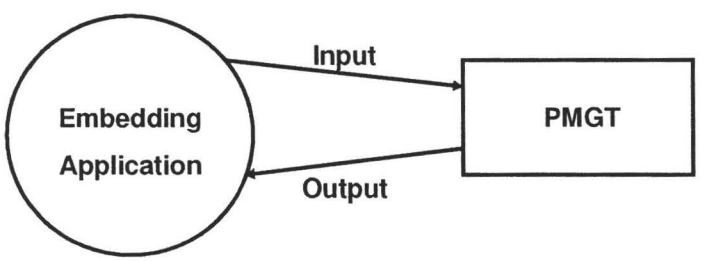

Figure A.1: System Context Diagram

Figure A.1 shows the context that PMGT will normally fit into. A circle represents an external entity outside the system, an embedding application in this case. The rectangle is the system itself. Arrows represent the data flows between them.

The *input*: $M^{\mathrm{IN}} \times I$

The *output*: M^{OUT}.

PMGT has the following function:

A mesh M^{IN} and some refining/coarsening instructions I are given. PMGT generates a refined/coarsened mesh M^{OUT} according to the instructions I.

A.3.2 User Characteristics

The target user group of PMGT includes both software designers, who intend to embed this library in their applications, and theoreticians, who are involved in parallel mesh generation. A user of PMGT is expected to be familiar with the notion/knowledge of mesh creation. PMGT is a library used by other applications. Therefore, users should not be novices in terms of software design. The prerequisite software design knowledge are equivalent to that of a senior undergraduate student in science or engineering who took an introductory course on programming. For example, they should be comfortable with compilation of the programming language in which PMGT is written, be familiar with embedding a library in their software, *etc.*

A.3.3 System Constraints

This system is intended to be built on the Shared Hierarchical Academic Research Computing Network (SHARCNET). SHARCNET is structured as a "cluster of clusters" across South Central Ontario, designed to meet the computational needs of researchers in a diverse number of research areas and to facilitate the development of leading-edge tools for high performance computing (HPC) grids.

Large production clusters, located at the Universities of Western Ontario, Guelph and McMaster, house over 400 HP/Compaq Alpha processors and large symmetric multiprocessor computers. Windsor and Wilfrid Laurier host smaller development clusters (8 processors), which enable researchers to develop and test code before moving to one of the larger clusters. A glance of SHARCNET systems is shown in Table A.1. Note that the network is constantly being updated. Detailed information can be found at SHARCNET (Last Access: January, 2006).

A.4 Specific System Requirements

This section describes the system requirements in detail. After the problem is clearly and unambiguously stated, some solution characteristics are specified.

System	Make	Type	CPUs	OS
bala	Compaq	Cluster	8	Red Hat Linux 7.2
cat	Unknown	Cluster	162	Red Hat Linux 8
goblin	Sun	Cluster	56	Fedora Core 2
hammerhead	Compaq	Cluster	112	Red Hat Linux 7.2
idra	Compaq SC	Cluster	128	Tru64
mako	HP	Cluster	16	Fedora Core 2
tiger	Compaq	Cluster	8	Red Hat Linux 7.2
typhon	Compaq	SMP	16	Tru64
wobbe	Unknown	Cluster	193	Red Hat Linux 8
TOTALS			699	

Table A.1: A Glance at the SHARCNET System

Non-functional requirements are also included in this section. The symbol := is used to indicate type definition. The notation for set building and expressions used in this section follows Gries and Schneider (1993). To define the notation, first let x be a list of dummies, t a type, R a predicate, E an expression, $*$ an operator, and P a predicate. Notation $\{x : t \mid R : E\}$ represents a set of values that result from evaluating $E[x := v]$ in the state for each value v in t such that $R[x := v]$ holds in that state. Expression $(*x : t \mid R : P)$ denotes the application of operator $*$ to the values P for all x in t for which range R is true.

A.4.1 Problem Description

The problems (goals) specified in this subsection represent ideal general models. The problems are simplified by introducing some assumptions, which are listed in Section A.4.2.

A.4.1.1 Background Overview

Many physical problems of importance to scientists and engineers are modeled as a set of Partial Differential Equations (PDEs). In most practical cases, it is necessary to solve the PDEs numerally. Numerical methods to solve PDEs frequently require that the domain of interest be divided into a mesh, which is a set of small, simple elements that cover the computational domain. In some applications, a single mesh is generated and used many times; in this case the processing time spent on mesh construction is not critical and a relatively slow, sequential algorithm suffices (Ruppert, 1993). However, some applica-

tions need adaptive meshing, which requires that the meshes be generated once and then modified many times. For instance, adaptive meshing is used for reliable Finite Element Analysis (FEA) using a posterrori error estimation (Zienkiewicz et al., 2005). The increased mesh interaction for adaptive meshing means an increased need for speed of managing the mesh data which suggest employing parallel processing techniques. Although generating a mesh using multiple processors is complicated, it can offer considerable speed-up over sequential processing. In addition, some FEA applications are implemented on multiple processors. If the adaptive mesh can be generated in multiple processors as well, the mesh data can remain on the local processors. Potentially, time to be used will be significantly reduced.

A.4.1.2 Goal Statements

There are two related goals for PMGT.

G1: Given a mesh M^{IN} and instructions I on how to refine the mesh, PMGT should generate a refined mesh M^{OUT} according to the instructions I.

G2: Given a mesh M^{IN} and instructions I on how to coarsen the mesh, PMGT should generate a coarsened mesh M^{OUT} according to the instructions I.

A.4.2 Solution Characteristics Specification

The goals stated in the last section are too general to achieve. In this section, the assumptions are specified first to reduce the scope of the software. Second, the theoretical models for the goals are described. Third, data definitions are given to assist with defining the theoretical models. Finally, the system behaviour is summarized.

A.4.2.1 Assumptions

A1: PMGT focuses on a 2D domain.

A2: The input and output meshes are bounded.

A3: The input and output meshes are unstructured.

A4: The input and output meshes are conformal.

A5: The elements of input and output meshes are triangles.

A6: The initial mesh is valid.

A.4.2.2 Theoretical Model

The theoretical models corresponding to the goals given in Section A.4.1 describes the relationship between the input mesh (M^{IN}) and the output mesh (M^{OUT}). The meshes are assumed to be embedded in a 2D space.

TM1: Refining Mesh

Input: M^{IN}: `MeshT`, I: `RCinstructionT`

Output: M^{OUT}: `MeshT`

The following behavior is specified:

- *Refined*(M^{OUT}, M^{IN})

That is, the output mesh is a refined version of the input mesh.

TM2: Coarsening Mesh

Input: M^{IN}: `MeshT`, I: `RCinstructionT`

Output: M^{OUT}: `MeshT`

The following behavior is specified:

- *Coarsened*(M^{OUT}, M^{IN})

That is, the output mesh is a coarsened version of the input mesh.

A.4.2.3 Data Definitions

The data definitions below are organized so that a definition listed in the beginning may be used to define a data item listed after it.

VertexT (D1): A vertex is represented by two real numbers, which are its x coordinate and y coordinate. More formally,
`VertexT` := tuple of ($x : \mathbb{R}$, $y : \mathbb{R}$).

EdgeT (D2): An edge is represented by a set of `VertexT`. More formally,
`EdgeT` := set of `VertexT`.

ValidEdge (D3): An edge is valid if the edge is a line segment (that is, the set has two elements). More formally,
ValidEdge: `EdgeT` $\rightarrow \mathbb{B}$
ValidEdge(*e*: `EdgeT`) $\equiv \#e = 2$

CellT (D4): A cell is represented by a set of *VertexT*. More formally,
`CellT` := set of `VertexT`

Area (D5): The area of a triangle whose apexes are elements of a cell. More formally,
Area: `CellT` $\rightarrow \mathbb{R}$
Area(*c*: `CellT`) $\equiv \Sigma v1, v2, v3$: `VertexT` $\mid v1 \in c \wedge v2 \in c \wedge v3 \in c$
$\wedge\, v1 \neq v2 \wedge v2 \neq v3 \wedge v3 \neq v1$:
$\frac{1}{12} * |v1.x * v2.y - v2.x * v1.y +$
$v2.x * v3.y - v3.x * v2.y +$
$v1.x * v3.y - v3.x * v1.y|$

ValidCell (D6): A cell is valid if the cell is a triangle (that is , the set has three elements) and the area of the triangle is greater than zero. More formally,
ValidCell: `CellT` $\rightarrow \mathbb{B}$
ValidCell(*c*: `CellT`) $\equiv \#c = 3 \wedge$ *Area*(*c*) ≥ 0

MeshT (D7): A mesh is represented by a set of cells. More formally,
`MeshT` := set of `CellT`.

OnEdge (D8): Checks if a vertex is on the line segment between two vertices (exclusive) of an edge. More formally,
OnEdge: `VertexT` $\times$ `EdgeT` $\rightarrow \mathbb{B}$
OnEdge(*v*: `VertexT`, *e*: `EdgeT`) $\equiv \exists\, v1$, *v*2: `VertexT` $\mid$
$v1 \in e \wedge v2 \in e \wedge v1 \neq v2 \wedge v \neq v1 \wedge v \neq v2$:
$(v1.x < v.x \leq v2.x \wedge$
$(v.y - v1.y)/(v.x - v1.x) = (v2.y - v1.y)/(v2.x - v1.x))$

BelongToCell (D9): Checks if an edge belongs to a cell. More formally,
BelongToCell: `VertexT` $\times$ `CellT` $\rightarrow \mathbb{B}$
BelongeToCell(*e*: `EdgeT`, *c*: `CellT`) $\equiv \forall\, v$: `VertexT` $\mid v \in e : v \in c$

Inside (D10): Checks if a point (of type VertexT) is inside of a cell. The *inside* checking is false if the point is on an edge of the cell or the point is a vertex of the cell. (The algorithm to check if a point is inside a polygon is from Blackpawn (Last Access: January, 2006).) More formally,

Inside: VertexT $\times$ CellT $\rightarrow \mathbb{B}$
Inside(v: VertexT, c: CellT) $\equiv \exists\ v1,\ v2,\ v3$: VertexT |
$v1 \in c \wedge v2 \in c \wedge v3 \in c \wedge v1 \neq v2 \wedge v2 \neq v3 \wedge v3 \neq v1$:
$((v.y - v1.y) * (v2.x - v1.x) - (v.x - v1.x) * (v2.y - v1.y)) *$
$((v.y - v2.y) * (v3.x - v2.x) - (v.x - v2.x) * (v3.y - v2.y)) > 0 \wedge$
$((v.y - v2.y) * (v3.x - v2.x) - (v.x - v2.x) * (v3.y - v2.y)) *$
$((v.y - v3.y) * (v1.x - v3.x) - (v.x - v3.x) * (v1.y - v3.y)) > 0$

Vertices (D11): A set of all vetices of the mesh. More formally,
Vertices: MeshT $\rightarrow$ set of VertexT
Vertices(m: MeshT) $\equiv \{v$: VertexT | $(\forall\ c$: CellT | $c \in m : v \in c) : v\}$

Edges (D12): A set of all edges of the mesh. More formally,
Edges: MeshT $\rightarrow$ set of *EdgeT*
Edges(m: MeshT) $\equiv \{v1,\ v2$: VertexT | $(\forall c$: CellT | $c \in m$:
$v1 \in c \wedge v2 \in c \wedge v1 \neq v2) : \{v1, v2\}\}$

BoundaryEdges (D13): A set of edges are boundary edges if they form a boundary of a mesh. More formally,
BoundaryEdges: MeshT $\rightarrow$ set of *EdgeT*
BoundaryEdges(m: MeshT) $\equiv \{b$: EdgeT | $b \in$ *Edges*(m) $\wedge$
$(\#\{c$: CellT | $c \in m \wedge$ *BelongToCell*(b, c): $c\} = 1)$: $b\}$

BoundaryVertices (D14): A set of boundary vertices of the mesh. More formally,
BoundaryVertices: MeshT $\rightarrow$ set of VertexT
BoundaryVertices(m: MeshT) $\equiv$
$\{v$: VertexT | $v \in$ *BoundaryEdges*(m): $v\}$

Bounded (D15): A mesh is bounded if the boundary edges form a closed polygon(all vertices of boundary edges belong to exactly two boundary edges). More formally,
Bounded: MeshT $\rightarrow \mathbb{B}$
Bounded(m: MeshT) $\equiv \forall v$: VertexT | $v \in$ *BoundaryVertices*(m):
$(\#\{e$: EdgeT | $e \in$ *BoundaryEdge*(m) $\wedge v \in e : e\} = 2)$

Conformal (D16): In 2D, a mesh is conformal if the intersection of any two cells is either a vertex or an edge or empty. More formally,
Conformal: MeshT $\rightarrow \mathbb{B}$
Conformal(m: MeshT) $\equiv \forall c1, c2$: CellT | $c1 \in m \wedge c2 \in m \wedge c1 \neq c2$:
$(\exists e$: EdgeT | $e \in$ *Eeges*(m): $(\exists v$: VertexT | $v \in$ *Vertices*(m):
$(c1 \cap c2 = e \vee c1 \cap c2 = v \vee c1 \cap c2 = \emptyset) \wedge (\neg$ *OnEdge*(v, e))))

NoInteriorIntersect (D17): NoInteriorIntersect is true if a point in space (of type VertexT) is inside only one cell of the mesh. More formally,
NoInteriorIntersect: `MeshT` $\rightarrow \mathbb{B}$
NoInteriorIntersect(*m*: `MeshT`) $\equiv \forall c1, c2$: `CellT` |
$c1 \in m \wedge c2 \in m \wedge c1 \neq c2$: (∀*v*: `VertexT` | *Inside(v, c1)*: $\neg$ *Inside(v,c2))*

ValidMesh (D18): A mesh is valid if the mesh is bounded, conformal, and any point is only inside one cell. More formally,
ValidMesh: `MeshT` $\rightarrow \mathbb{B}$
ValidMesh(*m*: `MeshT`) $\equiv$ (∀*e*: `EdgeT` | $e \in$ *Edges(m): ValidEdge(e))*
$\wedge$ (∀*c*: `CellT` | $c \in m$: *ValidCell(c))* $\wedge$
Bounded(m) $\wedge$ *Conformal(m)* $\wedge$ *NoInteriorIntersect(m)*

CoveringUp (D19): True if two meshes covering up each other, that is, if all endpoints of the boundary edges of one mesh are on the boundary edges or are end points of the boundary edges of another mesh. More formally,
CoveringUp: `MeshT` $\times$ `MeshT` $\rightarrow \mathbb{B}$
CoveringUp(*m1*, *m2*: `MeshT`) $\equiv \forall v1$, *v2*: `VertexT`, |
$v1 \in$ *BoundaryVertice(m1)* $\wedge$ $v2 \in$ *BoundaryVertices(m2)*:
(∃*b1*, *b2*: `EdgeT` | $b1 \in$ *BoundaryEdges(m1)* $\wedge b2 \in$ *BoundaryEdges(m2):*
(OnEdge(v1, b2) $\vee v1 \in b2) \wedge$ *(OnEdge(v2, b1)* $\vee v2 \in b1$))

InstructionT (D20): The type of instructions is defined as:
`InstructionT` := {`REFINE, COARSEN, NOCHANGE`}

CellInstructionT (D21): The type of instructions on a cell is defined as:
`CellInstructionT`:= tuple of (*cell*: `CellT`, *instr*: `InstructionT`)
(For each cell, there is an instruction for refining, coarsening, or nochange.)

RCinstructionT (D22): The type of instructions on a mesh is defined as:
`RCinstructionT`:= tuple of (*rORc*: `InstructionT`, *cInstr*: set of `CellInstructionT`)
(For each mesh, there is an instruction on whole mesh, and there are set of instruction on each cell.)

Refined (D23): True if a mesh M' is a refined mesh of a mesh M. More formally
Refined: `MeshT` $\times$ `MeshT` $\times$ `RCinstructionT` $\rightarrow \mathbb{B}$
Refined(*m'*, *m*: `MeshT`, *rc*: `RCinstructionT`) $\equiv$
$rc.rORc =$ `REFINE` $\wedge$ *ValidMesh(m)* $\wedge$ *ValidMesh(m')* $\wedge$
CoveringUp(m', m) $\wedge \#m' \geq \#m$

Coarsened (D24): True if a mesh M' is a coarsened mesh of a mesh M.
More formally
Coarsened: MeshT $\times$ MeshT $\times$ RCinstructionT $\rightarrow \mathbb{B}$
Coarsened(m', m: MeshT, *rc*: RCinstructionT) $\equiv$
$rc.rORc =$ COARSEN $\wedge$ *ValidMesh(m)* $\wedge$ *ValidMesh(m')* $\wedge$
CoveringUp(m', m) $\wedge \#m' \leq \#m$

A.4.2.4 System Behaviour

System Behaviour, shown through functional requirements, defines what the software should do. The functional requirements, as well as nonfunctional requirements in Section A.4.3, partially come from Smith and Chen (2004). Smith and Chen (2004) listed all requirements that are common for mesh generation systems. They also considered the difference between meshes in term of variabilities. However, the mesh generations analyzed by Smith and Chen (2004) are targeted at full FEA applications. PMGT only manages the geometric information about the mesh, not other FEA related information, such as boundary condition and material property. Hence, only commonalities that is meaningful for PMGT are selected. Variabilities with parameters of variation that are suitable for PMGT are also considered. Other part of the requirements are obtained from Dr. Smith.

New functional requirements, *RCInstruction* (F9) and *Help* (F16) are added. F9 is unique to PMGT and F16 facilities the non-functional requirements *Usability* (N6).

We specify both functional requirements and non-functional requirements in the tables. In each table, the field *Description* gives a brief description of this requirement. It tells what PMGT should do to fulfill this requirement. There are two potential sources, shown in the *Source* field, for each requirement. One source is from Smith and Chen (2004), and the other comes from Dr. Smith. If the requirement is from Smith and Chen (2004), then this field will show the commonality number, with a prefix C and the associated variability, shown by a prefix V. Where applicable, *Related Data Definitions* and *Related Theoretical Models* gives the numbers of related data definitions and the numbers of related theoretical models, respectively. These two field only appear for functional requirements. The *Binding Time* field either shows scope time or run time. *Scope time* means that this requirement is determined when the SRS is written. *Run time* means that this requirement is determined when the system is running. *History* records the time of creating and changing of the requirements.

Requirements Number	F1
Requirements Name	RefiningMesh
Description	PMGT should have capabilities for refining an existing mesh. $I.rORc = \texttt{REFINE} \wedge$ *Refined*(M^{OUT}, M^{IN})
Source	C1, V3
Related Data Definitions	D20, D22, D23
Related Theoretical Models	TM1
Binding Time	Scope time
History	Created – June, 2005. Modified – October, 2005. Change the name from "ImprovingMesh" to "RefiningMesh". Modified – October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number	F2
Requirements Name	CoarseningMesh
Description	PMGT should have capabilities for coarsening an existing mesh. $I.rORc = \texttt{COARSEN} \wedge$ *Coarsened*(M^{OUT}, M^{IN})
Source	C1, V3
Related Data Definitions	D20, D22, D24
Related Theoretical Models	TM2
Binding Time	Scope time
History	Created – October, 2006.

Requirements Number	F3
Requirements Name	RefiningOrCoarsening
Description	PMGT can either refine a given mesh to a refined mesh, or coarsen a mesh to a coarsened mesh. However, PMGT cannot do both refining and coarsening at the same time.
Source	C1, V3
Related Data Definitions	D20, D22, D23, D24
Related Theoretical Models	TM1, TM2
Binding Time	Run time
History	Created – October, 2006.

Requirements Number	F4
Requirements Name	MeshType
Description	The mesh generated by PMGT is unstructured.
Source	C1, V6
Related Data Definitions	N/A
Related Theoretical Models	N/A
Binding Time	Scope time
History	Created – June, 2005. Modified – October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number	F5
Requirements Name	ElmShape
Description	The shape of the elements in both input and output meshes are triangles. $\forall c1, c2$: `CellT` | $c1 \in M^{\mathrm{IN}} \wedge c2 \in M^{\mathrm{OUT}}$: $\#c1 = 3 \wedge \ Area(c1) > 0 \wedge$ $\#c2 = 3 \wedge \ Area(c2) > 0$
Source	C1, V9
Related Data Definitions	D4, D5
Related Theoretical Models	TM1, TM2
Binding Time	Scope time
History	Created – June, 2005. Modified – October, 2006. Take out the requirement for generating quadrilateral meshes. Modified – October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number	F6
Requirements Name	DomainDimension
Description	The computational domain is in 2D space.
Source	C1, V13
Related Data Definitions	N/A
Related Theoretical Models	TM1, TM2
Binding Time	Scope time
History	Created – June, 2005. Modified – October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number	F7
Requirements Name	Conformal
Description	Both input and output meshes are conformal. $Conformal(M^{\mathrm{IN}}) \wedge Conformal(M^{\mathrm{OUT}})$
Source	C1, V18
Related Data Definitions	D16
Related Theoretical Models	N/A
Binding Time	Scope time
History	Created - June, 2005. Modified - October, 2006. Take out the requirement for generating non-conformal meshes. Modified - October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number	F8
Requirements Name	InputDefinition
Description	The input of PMGT should be provided by the embedding application.
Source	C8
Related Data Definitions	D7, D20, D22
Related Theoretical Models	TM1, TM2
Binding Time	Scope time
History	Created - June, 2005. Modified–October 2005. Change the name from "Input" to "Input-Definition" to clarify that this requirements is about the source of the input. Modified - October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number **Requirements Name**	F9 RCInstruction
Description	The Instruction on how to refine/coarsen a mesh includes the instruction of whether to refine or coarsen the mesh and an individual instruction for each element of the the mesh to indicate refining, coarsening, or no change.
Source	Dr. Smith
Related Data Definitions	D20, D21, D22
Related Theoretical Models	TM1, TM2
Binding Time	Scope time
History	Created – June, 2005 Modified – October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number **Requirements Name**	F10 OutputStorage
Description	The output of PMGT is stored in memory or in files or in both memory and files.
Source	C12, Dr. Smith
Related Data Definitions	N/A
Related Theoretical Models	N/A
Binding Time	Run time
History	Created - June, 2005. Modified - October 2005. Change the name from "Output" to "OutputStorage" to clarify that this requirements is about the storage of the output. Modified - October 2006. Add the requirement of storing the output mesh in files or both memory and files. Modified - October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number **Requirements Name**	F11 VertexUniqueID
Description	Each vertex in the output file has a unique identifier.
Source	C2
Related Data Definitions	N/A
Related Theoretical Models	N/A
Binding Time	Scope time
History	Created - June, 2005. Modified - October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number	F12
Requirements Name	ElmUniqueID
Description	Each element in the output file has a unique identifier.
Source	C3
Related Data Definitions	N/A
Related Theoretical Models	N/A
Binding Time	Scope time
History	Created – June, 2005. Modified – October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number	F13
Requirements Name	ElmTopology
Description	The topology of an element in the output file is given by the connectivity of its set of vertices.
Source	C4
Related Data Definitions	N/A
Related Theoretical Models	N/A
Binding Time	Scope time
History	Created – June, 2005. Modified – October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number	F14
Requirements Name	OutElmOrder
Description	The element information in output files is listed in ascending order.
Source	C13, V34
Related Data Definitions	N/A
Related Theoretical Models	N/A
Binding Time	Scope time
History	Created – June, 2005. Modified – October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number	F15
Requirements Name	OutVertexOrder
Description	The vertex information, such as the coordinates, in output files is listed in ascending order.
Source	C14, V35
Related Data Definitions	N/A
Related Theoretical Models	N/A
Binding Time	Scope time
History	Created – June, 2005. Modified – October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

Requirements Number **Requirements Name**	F16 Help
Description	Helps on documenting the interface and the functionality of each function should be provided.
Source	Dr. Smith
Related Data Definitions	N/A
Related Theoretical Models	N/A
Binding Time	Scope time
History	Created - June, 2005. Modified - October 2005. Add the requirement of documenting functionality of each function. Modified - October, 2006. Field for "Related Data Definitions" and "Related Theoretical Models" are added.

A.4.3 Non-functional Requirements

All non-functional requirements listed in Smith and Chen (2004) are selected except for C16, which is solution tolerance, since a mesh refined/coarsened by different algorithms may have different solutions, but all of these solutions can still be valid. All potential output meshes are valid as long as the output meshes are covering/covered up meshes of the original mesh, and they are refined/coarsened according to the RCInstruction. The resulting mesh is difficult to measure in terms of *solution tolerance*. Three new non-functional requirements, which are *LookAndFeel* (N5), *Usability* (N6), and *Maintainability* (N7), are added. These requirements are mentioned in Lai (2004).

PMGT is difficult to validate. One reason is that the solution for refining/coarsening a mesh is unknown, as mentioned above. The other reason is that it is difficult to write validatable requirements, especially for nonfunctional requirements. For example, what is the proper way for specifying the requirement of *Usability* (N6) of PMGT? On the one hand, that *the software should easy to use* is not validatable. On the other hand, that *a person should be able to use the software in two days* is validatable. However, the measurement, *two days*, often lacks a justifiable rationale.

The approach to validate this kind of requirements are to compare it with other software with similar functionality. Phrases that are in italics and capitalized, such as *MANPROP*, represent constant defined in Section A.8.

Usually, these constants come from other applications with similar functionalities. For example, the *Usability* requirement of PMGT is presented as follows:

> This system should be easy to use. Users with the background specified in Section A.3.2 should take *LEARNTIME* to reproduce an example mesh, which is specified by the test case TC5 in the Appendix D.

First, more general requirement is given. Then, a suggestion to reproduce an example mesh is specified. The constant *LEARNTIME* is defined as the time to produce the same mesh for users with the same background using AOMD.

Requirements Number	N1
Requirements Name	Performance
Description	Refining/coarsening a mesh using multiple processors should be faster than when using a single processor. In addition, the performance of PMGT should be comparable with that of similar applications. The execution time to refine an example mesh, which is specified by the test case TC5 in the Appendix D.
Source	C15, V39
Binding Time	Scope time
History	Created – June, 2005.

Requirements Number	N2
Requirements Name	Precision
Description	The number of decimal digits should agree with the IEEE standard for floating-point numbers.
Source	C17, V41
Binding Time	Scope time
History	Created – June, 2005.

Requirements Number	N3
Requirements Name	Exception
Description	Run-time exception handling should check at least the following exceptions: division by zero, redundant vertices, redundant edges, redundant cells.
Source	C18, V42
Binding Time	Scope time
History	Created - June, 2005

Requirements Number	N4
Requirements Name	Portability
Description	PMGT should build on a platform with access to SHARCNET or on a the system that has similar architecture to SHARCNET. The memory capacity should be *MEMCAP*.
Source	C19, V37, V38
Binding Time	Scope time
History	Created - June, 2005

Requirements Number	N5
Requirements Name	LookAndFeel
Description	PMGT should follow the programming conventions of the language in which the application is coded in.
Source	Dr. Smith
Binding Time	Scope time
History	Created - June, 2005

Requirements Number	N6
Requirements Name	Usability
Description	This system should be easy to use. Users with the background specified in Section A.3.2 should take *LEARNTIME* to reproduce an example mesh, which is specified in the Appendex D.
Source	Dr. Smith
Binding Time	Scope time
History	Created - June, 2005

Requirements Number	N7
Requirements Name	Maintainability
Description	The system should be developed in the way that the effort spent to maintain the system or to add in features would be minimum. The redevelopment time to add a new algorithm to coarsen meshes in PMGT should be *MANPROP*.
Source	Dr. Smith
Binding Time	Scope time
History	Created - June, 2005

A.5 Other System Issues

This section includes some other supporting information that might contribute to the success or failure of the system development. The following factors are considered:

- Open issues are statements of factors that are uncertain and might make significant difference to the system.
- Off-the-shell solutions are existing systems and/or components bought or borrowed. They could be the potential solutions.
- Waiting rooms provide a blueprint of how the system will be extended.

A.5.1 Open Issues

There are no open issues for PMGT at this stage.

A.5.2 Off-the-shelf Solutions

The following programs may be used in PMGT.

- AOMD: a mesh management library (or database) that is able to provide a variety of services for mesh users (SCOREC, Last Access: January, 2006).

A.5.3 Waiting Rooms

Here, we list the possible changes that can affect the extension of the system. These changes are related to the assumptions specified in Section A.4.2.

1. PMGT may produce both structured and unstructured meshes.
2. PMGT may produce both conformal and nonconformal meshes.
3. The elements of input and output mesh may be of a shape other than triangles.
4. The system may deal with invalid input mesh.
5. The system may accommodate a mixed mesh.
6. The system may accommodate a hybrid mesh.
7. The system may deal with a 3D problem domain.

A.6 Traceability Matrix

The traceability matrix defined in this section gives a big picture of the associations among goals, assumptions, data definitions, theoretical models, and functional requirements. Goals are ideal general models. After assumptions are applied, these goals are restricted to problems that can be solved by PMGT. Data definitions and theoretical models are used to describe the requirements. The matrix is too big to fit one page. For the sake of clarity, it is split into three parts in five tables, which are Table A.2, Table A.3, Table A.4, Table A.5, and Table A.6. In addition, only items that have a relation with items

in the same part are listed. If there is a ✓in a cell, it means that if the goal, or the assumption, or the theoretical model, or the data definition, or the requirement in the corresponding column changes, the assumption, or the data definition, or the theoretical model, or the requirement in the corresponding row should also change.

A.7 List of Possible Changes in the Requirements

The system might evolve to accommodate the following changes in the future. These changes will add additional goals to the software library.

1. The input of PMGT may include material properties.
2. The input of PMGT may include boundary conditions.

A.8 Values of Auxiliary Constants

The constants given in this section are used to validate some nonfunctional requirements. The compatible software chosen is AOMD. However, other software can also be used as long as the other software has the required functionalities to validate the given requirement.

LEARNTIME The time that reproduce the same example as that specified in nonfunctional requirement N6 using AOMD.

MANPROP The redevelopment time to add the same algorithm as that specified in the nonfunctional requirement N7, using AOMD. If the algorithm is already in AOMD, the the time that AOMD took to add it.

RSPTIME The execution time to refine the same mesh as that specified in nonfunctional requirement N1 using AOMD.

MEMCAP The typical memory capacity of a machine on SHARCNET.

	G1	G2	A1	A2	A3	A4	A5	A6	TM1	TM2
A1	✓	✓	✓							
A2	✓	✓		✓						
A3	✓	✓			✓					
A4	✓	✓				✓				
A5	✓	✓	✓				✓			
A6	✓	✓						✓		
D1	✓	✓	✓							
D2	✓	✓								
D3	✓	✓								
D4	✓	✓	✓				✓			
D5	✓	✓	✓				✓			
D6	✓	✓	✓				✓			
D7	✓	✓								
D8	✓	✓	✓							
D9	✓	✓	✓							
D10	✓	✓	✓				✓			
D11	✓	✓								
D12	✓	✓	✓					✓		
D13	✓	✓	✓							
D14	✓	✓	✓							
D15	✓	✓	✓	✓						
D16	✓	✓	✓			✓				
D17	✓	✓	✓	✓						
D18	✓	✓								
D20	✓	✓								
D21	✓	✓								
D22	✓	✓								
D19	✓	✓	✓	✓						
D23	✓									
D24		✓								
TM1	✓		✓						✓	
TM2		✓	✓							✓

Table A.2: Traceability Matrix (PART I): Goals, Assumptions, Theoretical Models, Data Definitions, and Requirements (I)

	G1	G2	A1	A2	A3	A4	A5	A6	TM1	TM2
F1	✓								✓	
F2		✓								✓
F3	✓	✓							✓	✓
F4					✓					
F5	✓	✓	✓				✓		✓	✓
F6	✓	✓	✓						✓	✓
F7	✓	✓				✓				
F8	✓	✓							✓	✓
F9	✓	✓							✓	✓
F10	✓	✓								
F16	✓	✓								

Table A.3: Traceability Matrix (PART I): Goals, Assumptions, Theoretical Models, Data Definitions, and Requirements (II)

	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12
D1	✓											
D2	✓	✓										
D3		✓	✓									
D4	✓			✓								
D5	✓				✓							
D6				✓	✓	✓						
D7				✓			✓					
D8	✓	✓						✓				
D9	✓	✓		✓					✓			
D10	✓			✓						✓		
D11	✓			✓			✓				✓	
D12	✓	✓		✓			✓					✓
D13		✓		✓					✓			✓
D14	✓						✓					
D15	✓	✓					✓					
D16	✓	✓		✓			✓				✓	✓
D17	✓			✓			✓			✓		
D18			✓			✓	✓					
D19	✓	✓					✓	✓				
D20												
D21				✓								
D22												
D23												
D24												
TM1							✓					
TM2							✓					
F1												
F2												
F3												✓
F5				✓		✓						
F7												
F8							✓					
F9												

Table A.4: Traceability Matrix (PART II): Data Definitions and Requirements (I)

	D13	D14	D15	D16	D17	D18	D19	D20	D21	D22	D23	D24
D13	✓											
D14	✓	✓										
D15	✓	✓	✓									
D16				✓								
D17					✓							
D18			✓	✓	✓	✓						
D19	✓	✓					✓					
D20								✓				
D21								✓	✓			
D22								✓	✓	✓		
D23						✓	✓	✓		✓	✓	
D24						✓	✓	✓		✓		✓
TM1										✓	✓	
TM2										✓		✓
F1								✓		✓	✓	
F2								✓		✓		✓
F3								✓		✓	✓	✓
F5												
F7				✓								
F8								✓		✓		
F9								✓	✓	✓		

Table A.5: Traceability Matrix (PART II): Data Definitions and Requirements (II)

	F1	F2	F6	F8	F10	N6
F3	✓	✓				
F5			✓			
F9				✓		
F11					✓	
F12					✓	
F13					✓	
F14					✓	
F15					✓	
F16						✓

Table A.6: Traceability Matrix (PART III): Requirements

Appendix B

Module Guide for a Parallel Mesh Generation Toolbox

B.1 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team. The basic principle of the decomposition used here is the information hiding principle (Parnas et al., 1984). According to Parnas et al. (1984),

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is used in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) for the PMGT was developed. The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure of the PMGT and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it is can be used to check for consistency, feasibility and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as described in the following. Section B.2 lists the anticipated and unlikely changes of the software requirements. Section B.3 summarizes the module decomposition that was constructed according to the likely changes. Section B.4 specifies the connections between the software requirements and the modules. Section B.5 gives a detailed description of the modules. Section B.6 includes two traceability matrices. One

checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section B.7 describes the use relation between modules.

B.2 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section B.2.1, and unlikely changes are listed in Section B.2.2.

B.2.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

AC1: The data structure and algorithms for implementing the virtual memory of the system.

AC2: The data structure and algorithms for implementing the interface between the file and the system.

AC3: The data structure and algorithms for implementing the interface between the keyboard and the system.

AC4: The data structure and algorithms for screen display.

AC5: The format and structure of the initial input mesh.

AC6: The format and structure of the output mesh.

AC7: The mechanisms for validating the input and output meshes.

AC8: The data structure of a vertex.

AC9: The data structure of an edge.

AC10: The data structure of a cell.

AC11: The data structure of a mesh.

AC12: The algorithms for refining a mesh.

AC13: The algorithms for coarsening a mesh.

AC14: The shape of a cell, which is initially assumed to be a triangular.

B.2.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

UC1: Input/Output devices (Input: File and/or Keyboard, Output: File, Memory, and/or Screen).

UC2: There will always be a source of input data external to the PMGT software.

UC3: Output data are displayed to the output device.

UC4: The goal of the system is refining or coarsening a mesh.

UC5: The type of the mesh is unstructured.

UC6: The representation of an edge is a set of vertices.

UC7: The representation of a cell is a set of vertices.

UC8: A Cartesian coordinate system is used.

B.3 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table B.1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: Virtual Memory Module

M2: File Read/Write Module

M3: Keyboard Input Module

M4: Screen Display Module

M5: Input Format Module

M6: Output Format Module

M7: Service Module

M8: Vertex Module

M9: Edge Module

M10: Cell Module

M11: Mesh Module

M12: Refining Module

M13: Coarsening Module

Note that M1, M2, M3 and M4 are commonly used modules and are already implemented by the operating system. They will not need to be implemented again for PMGT.

B.4 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table B.2. However, some connections are not obvious. The explanation below has the purpose of making these connections clear. The software requirements are documented in the SRS. They are also listed starting on page 162 for convenience.

The functionalities of refining a mesh (F1), and coarsening a mesh (F2) are achieved directly by M12 and M13, respectively. The functional requirement *MeshType* (F4) is related to the representation of mesh, which is contained in M9, M10, and M11. The algorithms for refining (M12) and coarsening (M13) also depend on the *MeshType* requirement. Another connection worth mentioning relates to the *DomainDimension* requirement (F6). All geometric information for the mesh, including dimension information, is stored in M8. Algorithms in M12 and M13 also relate to the dimension of the domain.

Level 1	**Level 2**	**Level 3**	**Level 4**
Hardware-Hiding Module	Extended Computer Module	Virtual Memory Module	
		File Read/Write Module	
	Device Interface Module	Keyboard Input Module	
		Screen Display Module	
Behavior-Hiding Module	Input Format Module		
	Output Format Module		
	Service Module		
Software Decision Module	Mesh Data Module	Entity Module	Vertex Module
			Edge Module
			Cell Module
		Mesh Module	
	Algorithm Module	Refining Module	
		Coarsening Module	

Table B.1: Module Hierarchy

Some nonfunctional requirements, such as *Performance* (N1) and *Maintainability* (N7), are related to the overall quality of the system. These qualities depend on the implementation of all of the modules. The *Precision* requirement depends on modules related to calculation, which are the module M8, M9, M10, M11, M12 and M13.

B.5 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. *PMGT* means the module will be implemented by the PMGT soft-

ware. Only leaf modules in the hierarchy have to be implemented. If a dash (-) is shown, this means that the module is not a leaf and will not have to be implemented. Whether or not this module is implemented depends on the programming language selected. This decomposition is inspired by Chen (2003). The decomposition of the mesh data module is partly based on ElSheikh et al. (2004). One difference between the current design and ElSheikh et al. (2004) is that ElSheikh et al. (2004) has an explicit module for incidence and adjacency information. However, it is believed that where and how to store this information is an implementation decision that should be abstracted away at the design stage.

B.5.1 Hardware-Hiding Module

Secrets: The data structure and algorithm used to implement the virtual hardware.

Services: Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

Implemented By: -

B.5.1.1 Extended Computer Module

Secrets: The number of processors, the instruction set of the computer, and the computer's capacity for performing concurrent operations.

Services: Provides an instruction set including the operations on application-independent data types, sequence control operations, and general I/O operations.

Implemented By: -

B.5.1.1.1 Virtual Memory Module (M1)

Secrets: The hardware addressing methods for data and instructions in real memory.

Services: Presents a uniformly addressable virtual memory.

Implemented By: OS

B.5.1.1.2 File Read Write Module (M2)

Secrets: The data structure and algorithms for implementing the interface between the file and the system.

Services: Provides an interface between the storage of the system and the IO devices.

Implemented By: OS

B.5.1.2 Device Interface Module

Secrets: Characteristics of the present devices not likely to be shared by replacement devices.

Services: Provides virtual devices to be used by the rest of software.

Implemented By: -

B.5.1.2.1 Keyboard Input Module (M3)

Secrets: The data structure and algorithms for implementing the interface between the keyboard and the system.

Services: Retrieves the user inputs from the keyboard and communicates the information with other parts of the system.

Implemented By: OS

B.5.1.2.2 Screen Display Module (M4)

Secrets: The data structure and algorithms to display graphics and text on the screen.

Services: Provides an interface between the system and the screen so the system can display information on the screen through the use of programs in the module.

Implemented By: OS

B.5.2 Behavior-Hiding Module

Secrets: The contents of the required behaviors.

Services: Includes programs that provide externally visible behavior of the system as specified in the software requirements specification (SRS) documents. This module serves as a communication layer between the hardware-hiding module and the software decision module. The programs in this module will need to change if there are changes in the SRS.

Implemented By: -

B.5.2.1 Input Format Module (M5)

Secrets: The format and structure of the initial input mesh.

Services: Converts the input mesh to the data structured used in PMGT.

Implemented By: PMGT

B.5.2.2 Output Format Module (M6)

Secrets: The format and structure of the output mesh.

Services: Converts the output mesh to an output file.

Implemented By: PMGT

B.5.2.3 Service Module (M7)

Secrets: The algorithm for validating meshes.

Services: Checks if the input and output meshes are valid.

Implemented By: PMGT

B.5.3 Software Decision Module

Secrets: The design decision based on mathematical theorems, physical facts, or programming considerations. The secrets of this module are *not* described in the SRS.

Services: Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

Implemented By: -

B.5.3.1 Entity Module

Secrets: The data structure of a mesh entity, including vertex, edge, and cell.

Services: Stores the complete mesh information generated, and also provides programs to import and export the mesh information.

Implemented By: -

B.5.3.1.1 Vertex Module (M8)

Secrets: The data structure of a vertex.

Services: Stores the complete vertex information generated and provides programs to import and export the vertex information. The operations on vertices are also included in this module.

Implemented By: PMGT

B.5.3.1.2 Edge Module (M9)

Secrets: The data structure of an edge.

Services: Stores the complete edge information generated and provides programs to import and export the edge information. The operations on edges are also included in this module.

Implemented By: PMGT

B.5.3.1.3 Cell Module (M10)

Secrets: The data structure of a cell.

Services: Stores the complete cell information generated and provides programs to import and export the cell information. The operations on cells are also included in this module.

Implemented By: PMGT

B.5.3.1.4 Mesh Module (M11)

Secrets: The data structure of a mesh.

Services: Stores the complete mesh information generated and provides programs to import and export the cell information. The operations on meshes are also included in this module.

Implemented By: PMGT

B.5.3.2 Mesh Algorithm Module

Secrets: Algorithms for refining and coarsening a mesh.

Services: Refining and coarsening a mesh.

Implemented By: -

B.5.3.2.1 Refining Module (M12)

Secrets: Algorithms for refining a mesh.

Services: Refining a mesh.

Implemented By: MPGT

B.5.3.2.2 Coarsening Module (M13)

Secrets: Algorithms for coarsening a mesh.

Services: Coarsening a mesh.

Implemented By: MPGT

B.6 Traceability Matrix

A traceability matrix can be used for checking the completeness of the current design. In this section, there are two matrices, the traceability matrix for requirements and the traceability matrix for anticipated changes. The module names and their corresponding numbers are can be found in Section B.3

B.6.1 Traceability Matrix for Requirements

The traceability matrix in Table B.2 makes a connection between the modules and the requirements. Modules are listed in the first row and requirements are listed in the first column. If a module, say A, satisfies a requirement, say B, and A is in j-th column and B in i-th row, then there is a check mark ✓in the cell of the i-th row and the j-th column. There is a special column "Doc." It represents the documentation of PMGT. the "Doc" entry is used to fulfill the requirement *Help* (F16). The names of the requirements and their corresponding numbers are listed below for convenience.

F1: RefiningMesh

F2: CoarseningMesh

F3: RefiningOrCoarsening

F4: MeshType

F5: ElmShape

F6: DomainDimension

F7: Conformal

F8: InputDefinition

F9: RCInstruction

F10: OutputStorage

F11: VertexUniqueID

F12: ElmUniqueID

F13: ElmTopology

F14: OutElmOrder

F15: OutVertexOrder

F16: Help

N1: Performance

N2: Precision

N3: Exception

N4: Portability

N5: LookAndFeel

N6: Usability

N7: Maintainability

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	Doc
F1												✓		
F2													✓	
F3												✓	✓	
F4									✓	✓	✓	✓	✓	
F5							✓			✓				
F6								✓				✓	✓	
F7							✓					✓	✓	
F8		✓			✓									
F9												✓	✓	
F10	✓	✓				✓								
F11						✓								
F12						✓								
F13						✓								
F14						✓								
F15						✓								
F16														✓
N1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
N2					✓	✓	✓	✓	✓	✓	✓	✓	✓	
N3	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	
N4	✓	✓	✓	✓	✓	✓	✓							
N5					✓	✓	✓	✓	✓	✓	✓	✓	✓	
N6			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
N7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table B.2: Traceability Matrix: Modules and Requirements

B.6.2 Traceability Matrix for Anticipated Changes

The traceability matrix in Table B.3 illustrates the relationship between modules and anticipated changes listed in Section B.2. If there is a ✓in an entry of the matrix, the change specified in that row is hidden in the module of the corresponding column.

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13
AC1	✓												
AC2		✓											
AC3			✓										
AC4				✓									
AC5					✓								
AC6						✓							
AC7							✓						
AC8								✓					
AC9									✓				
AC10										✓			
AC11											✓		
AC12												✓	
AC13													✓
AC14							✓			✓			

Table B.3: Traceability Matrix: Modules and Anticipated Changes

B.7 Use Hierarchy between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure B.1 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

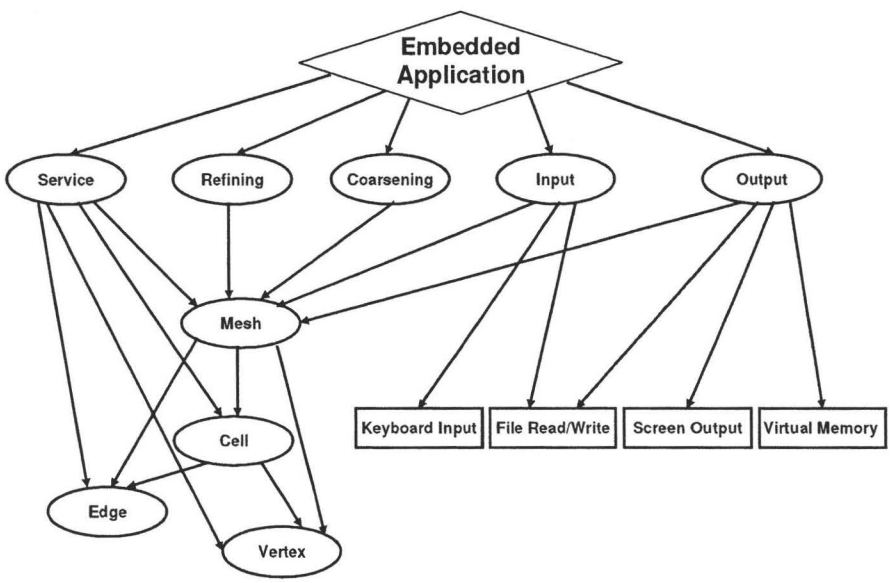

Figure B.1: Use Hierarchy among Modules

Appendix C

Module Interface Specification for a Parallel Mesh Generation Toolbox

C.1 Introduction

One of the advantages of decomposing the system into modules is that each module can be developed independently. However, the secret and services of each module does not provide enough information for parallel coding. A document specifying the interface of each module, called the Module Interface Specification (MIS), is needed. An MIS of a particular module is not only used as a guide by the programmers that are responsible for coding this module, but also by programmers that will use this module. An MIS is abstract because it describes *what* the module will do, but not *how* to do it.

This MIS describes the services of the corresponding modules specified in the document "Module Guide for a Mesh Generator." A state machine MIS is used. Note that some of the modules have multiple projections. In this case, variables listed in section *state variables* give the format of all states for all of the created objects. The idea of multiple projection is also used in Bauer (1995). By using projections, the change of state variables is applicable to the particular object associated with this module. In this system, this particular kind of modules includes the module Vertex, Edge, Cell, and Mesh.

The rest of the document is organized as follows. Section C.2 describes the MIS template used in this document. Section C.3 copies the module hierarchy from *Module Guide* document for convenience. Section C.4, Section C.5, Section C.6, Section C.7, Section C.8, Section C.11, Section C.12 give the MISs for the Vertex Module, Edge Module, Cell Module, Mesh Module, Service Module, Refining Module, and Coarsening Module, respectively.

C.2 Template

This section gives the template used in this document. This template is modified version of the MIS template presented in Ghezzi et al. (2003) and Hoffman and Strooper (1999). According to this template, each module is modeled as a finite state machine. It has a set of state variables, inputs, outputs, and transitions. In the case that an exception conditions become true, an exception is raised by the associated access program. If an access program has an output, then *Output* is specified. If an access program changes states variables, a *Transition* is specified. The inputs of the access program are listed as arguments.

The discrete mathematics notation used here follows that introduced by Gries and Schneider (1993). This notation is explained in the SRS. A dot notation is used in two cases. One is for referring to a field in a tuple, and the

other is for referencing the access program of a module.

The whole template is composed of four parts. First, the name of the module is given. Second, constants, data types, and access programs that are used by this module, but defined outside of this module, are listed. Third, the syntax of the interface is specified. Finally, the semantics of the interface is described. The template is described in the rest of this section.

C.2.1 Module Name

If "(MP)" is appended to the name of the module, it means that this module has multiple projections.

C.2.2 Uses

This section lists constants, data types, and access programs that are defined outside of this module. The format of each imported item is specified after each header.

C.2.2.1 Imported Constants

Uses ⟨ *module name* ⟩ **Imports** ⟨ *resource constants list* ⟩

C.2.2.2 Imported Data Types

Uses ⟨ *module name* ⟩ **Imports** ⟨ *resource data type list*⟩

C.2.2.3 Imported Access Programs

Uses ⟨ *module name* ⟩ **Imports** ⟨ *resource access program list*⟩

C.2.3 Interface Syntax

This section defines the syntax of the module interface. The interface indicates the services that the module provides. Other modules can only access this module through this interface. Other information inside the module is the secret that it hides from other modules. Changing the internal design of a module will not affect the way that other modules use this module. The format of each exported items is specified after each header.

C.2.3.1 Exported Constants

constant name : type of the constant

C.2.3.2 Exported Data Types

data type name := structure of the data type

C.2.3.3 Exported Access Programs

The exported access programs are listed in the tabular format shown below. In this software, exceptions are handled inside the access routine by displaying error messages and terminating the program.

Routine Name	Input	Output	Exceptions

C.2.4 Interface Semantics

The semantics of the interface is introduced in this section. The components of this section include state variables, state invariants, access program semantics, *etc.*

C.2.4.1 State Variables

This section lists the state variables in the format of *variable name*: *type*

C.2.4.2 Assumption

Any assumption about this module are specified here.

C.2.4.3 Invariant

Predicates that should always hold before and after each access routine in the module.

C.2.4.4 Access Program Semantics

This section includes possible exceptions, possible outputs, and possible transitions. The contents of this section should be as formal as possible. When necessary and appropriate, an English explanation is included to help readers understand the meaning of some the mathematical notations.

C.2.4.5 Local Functions

Functions used to facilitate the expression of the interface semantics.

C.2.4.6 Local Data Types

Data types used to facilitate the expression of the interface semantics.

C.2.4.7 Local Constants

Constants used to facilitate the expression of interface semantics.

C.2.4.8 Considerations

Other issues related to the MIS of this module, but not covered in the other parts of the document.

C.3 Module Decomposition

PMGT is decomposited into the modules listed in Table C.1. Note that only the leaf modules are implemented. The *Virtual Memory Module*, *File Read/Write Module*, *Keyboard Input Module*, and *Screen Display Module* are implemented by the operating system and programming language libraries. More information on the modular decomposition of the PMGT can be found in the MG document.

C.4 MIS of Vertex Module

C.4.1 Module Name: Vertex (MP)

C.4.2 Uses

C.4.2.1 Imported Constants

None

C.4.2.2 Imported Data Types

None

Level 1	**Level 2**	**Level 3**	**Level 4**
Hardware-Hiding Module	Extended Computer Module	Virtual Memory Module	
		File Read/Write Module	
	Device Interface Module	Keyboard Input Module	
		Screen Display Module	
Behavior-Hiding Module	Input Format Module		
	Output Format Module		
	Service Module		
Software Decision Module	Mesh Data Module	Entity Module	Vertex Module
			Edge Module
			Cell Module
		Mesh Module	
	Algorithm Module	Refining Module	
		Coarsening Module	

Table C.1: Module Hierarchy

C.4.2.3 Imported Access Programs

None

C.4.3 Interface Syntax

C.4.3.1 Exported constants

None

C.4.3.2 Exported Data Types

`VertexT` := tuple of $(x : \mathbb{R},\ y : \mathbb{R})$

C.4.3.3 Exported Access Programs

The exported access programs for the vertex module are listed in Table C.2.

Routine Name	Input	Output	Exceptions
initVertex	$\mathbb{R}$, $\mathbb{R}$		
getVertex		`VertexT`	

Table C.2: Exported Access Programs of the Vertex Module

C.4.4 Interface Semantics

C.4.4.1 State Variables

$x : \mathbb{R}$
$y : \mathbb{R}$

C.4.4.2 Invariant

None

C.4.4.3 Assumptions

initVertex() is called before any other access routine.

C.4.4.4 Access Program Semantics

C.4.4.4.1 initVertex($x1 : \mathbb{R},\ y1 : \mathbb{R}$)

- **Transition**
  $x := x1$
  $y := y1$

C.4.4.4.2 getVertex()

- **Output**
  (x, y)

C.4.4.5 Local Functions

None

C.4.4.6 Local Data Types

None

C.4.4.7 Local Constants

None

C.4.4.8 Considerations

None

C.5 MIS of Edge Module

C.5.1 Module Name: Edge (MP)

C.5.2 Uses

C.5.2.1 Imported Constants

None

C.5.2.2 Imported Data Types

Uses *Vertex Module* **Imports** `VertexT`

C.5.2.3 Imported Access Programs

None

C.5.3 Interface Syntax

C.5.3.1 Exported constants

None

C.5.3.2 Exported Data Types

`EdgeT` := set of `VertexT`

C.5.3.3 Exported Access Programs

The exported access programs for the Edge module are listed in Table C.3.

Routine Name	Input	Output	Exceptions
initEdge	`VertexT, VertexT`		EqualVertices
getEdge		`EdgeT`	

Table C.3: Exported Access Programs of the Edge Module

C.5.4 Interface Semantics

C.5.4.1 State Variables

e: set of `VertexT`

C.5.4.2 Invariant

$\#e = 2$

C.5.4.3 Assumptions

initEdge() is called before any other access routine.

C.5.4.4 Access Program Semantics

None

C.5.4.4.1 initEdge(*start*: `VertexT`, *end*: `VertexT`)

- **Exception**
  $start = end \Longrightarrow$ EqualVertices
- **Transition**
  $e: = \{start, end\}$

C.5.4.4.2 getEdge()

- **Output**
  e

C.5.4.5 Local Functions

None

C.5.4.6 Local Data Types

None

C.5.4.7 Local Constants

None

C.5.4.8 Considerations

None

C.6 MIS of Cell Module

C.6.1 Module Name: Cell (MP)

C.6.2 Uses

C.6.2.1 Imported Constants

None

C.6.2.2 Imported Data Types

Uses *Vertex Module* **Imports** `VertexT`

C.6.2.3 Imported Access Programs

None

C.6.3 Interface Syntax

C.6.3.1 Exported constants

None

C.6.3.2 Exported Data Types

`CellT` := set of `VertexT`

C.6.3.3 Exported Access Programs

The exported access programs for the cell module are listed in Table C.4.

Routine Name	Input	Output	Exceptions
initCell	`VertexT, VertexT, VertexT`		EqualVertices
getCell		`CellT`	

Table C.4: Exported Access Programs of the Cell Module

C.6.4 Interface Semantics

C.6.4.1 State Variables

c: set of `VertexT`

C.6.4.2 Invariant

$\#c = 3$

C.6.4.3 Assumptions

initCell() is called before any other access routine.

C.6.4.4 Access Program Semantics

None

C.6.4.4.1 initCell($v1$: `VertexT`, $v2$: `VertexT`, $v3$: `VertexT`)

- **Exception**
  $v1 = v2 \vee v2 = v3 \vee v3 = v1 \Longrightarrow$ EqualVertices
- **Transition**
  $c := \{v1, v2, v3\}$

C.6.4.4.2 getCell()

- **Output**
  c

C.6.4.5 Local Functions

None

C.6.4.6 Local Data Types

None

C.6.4.7 Local Constants

None

C.6.4.8 Considerations

None

C.7 MIS of Mesh Module

C.7.1 Module Name: Mesh (MP)

C.7.2 Uses

C.7.2.1 Imported Constants

None

C.7.2.2 Imported Data Types

Uses *Vertex Module* **Imports** `VertexT`
Uses *Edge Module* **Imports** `EdgeT`
Uses *Cell Module* **Imports** `CellT`

C.7.2.3 Imported Access Programs

None

C.7.3 Interface Syntax

C.7.3.1 Exported constants

None

C.7.3.2 Exported Data Types

`MeshT` := set of `CellT`

C.7.3.3 Exported Access Programs

The exported access programs for the mesh module are listed in Table C.5.

Routine Name	Input	Output	Exceptions
initMesh			
getMesh		`MeshT`	
numOfCells		$\mathbb{N}$	
addCell	`CellT`		CellExist
deleteCell	`CellT`		CellNotExist
onEdge	`VertexT, EdgeT`	$\mathbb{B}$	
belongToCell	`EdgeT , CellT`	$\mathbb{B}$	
inside	`VertexT , CellT`	$\mathbb{B}$	
vertices		set of `VertexT`	
edges		set of `EdgeT`	
boundaryEdges		set of `EdgeT`	
boundaryVertices		set of `VertexT`	

Table C.5: Exported Access Programs of the Mesh Module

C.7.4 Interface Semantics

C.7.4.1 State Variables

m: set of `CellT`

C.7.4.2 Invariant

$\#m \geq 0$

C.7.4.3 Assumptions

initCell() is called before any other access routine.

C.7.4.4 Access Program Semantics

C.7.4.4.1 initMesh()

- **Transition**
  $m := \emptyset$

C.7.4.4.2 getMesh()

- **Output**
  m

C.7.4.4.3 numOfCells()

- **Output**
  $\#m$

C.7.4.4.4 addCell(c: `CellT`)

- **Exception**
  $c \in m \Longrightarrow$ CellExist

- **Transition**
  $m := m \cup \{c\}$

C.7.4.4.5 deleteCell(c: `CellT`)

- **Exception**
  $c \notin m \Longrightarrow$ CellNotExist

- **Transition**
  $m := m \backslash \{c\}$

C.7.4.4.6 onEdge(v: `VertexT`, e: `EdgeT`)

- **Description**
  Returns true if a vertex v is on the line segment between two vertices (exclusive) of the edge e.

- **Output**
  $\exists v1, v2$: `VertexT` |
  $v1 \in e \wedge v2 \in e \wedge v1 \neq v2 \wedge v \neq v1 \wedge v \neq v2$:
  $(v1.x < v.x \leq v2.x \wedge$
  $(v.y - v1.y)/(v.x - v1.x) = (v2.y - v1.y)/(v2.x - v1.x))$

C.7.4.4.7 belongToCell(e: `EdgeT`, c: `CellT`)

- **Description**
  Returns true if an edge e belongs to a cell c.

- **Output**
  $\forall v$: `VertexT` | $v \in e : v \in c$

C.7.4.4.8 inside(v: VertexT, c: CellT)

- **Description**
  Returns true if a vertex v is inside a cell c. (The algorithm is adopt from Franklin (Last Access: January, 2006).)

- **Output**
  $\exists v1, v2, v3$: VertexT |
  $v1 \in c \wedge v2 \in c \wedge v3 \in c \wedge v1 \neq v2 \wedge v2 \neq v3 \wedge v3 \neq v1 :$
  $((v.y - v1.y) * (v2.x - v1.x) - (v.x - v1.x) * (v2.y - v1.y)) *$
  $((v.y - v2.y) * (v3.x - v2.x) - (v.x - v2.x) * (v3.y - v2.y)) > 0 \wedge$
  $((v.y - v2.y) * (v3.x - v2.x) - (v.x - v2.x) * (v3.y - v2.y)) *$
  $((v.y - v3.y) * (v1.x - v3.x) - (v.x - v3.x) * (v1.y - v3.y)) > 0$

C.7.4.4.9 vertices()

- **Description**
  Returns the set of all the vertices of the mesh.

- **Output**
  $\{v$: VertexT | $(\forall c$: CellT | $c \in m : v \in c) : v\}$

C.7.4.4.10 edges()

- **Description**
  Returns the set of all the edges of the mesh

- **Output**
  $\{v1, v2$: VertexT | $(\forall c$: CellT | $c \in m : v1 \in c \wedge v2 \in c \wedge v1 \neq v2) :$
  $\{v1, v2\}\}$

C.7.4.4.11 boundaryEdges()

- **Description**
  Returns a set of boundary edges of the mesh

- **Output**
  $\{b$: EdgeT | $b \in$ Edges() $\wedge$
  $(\#\{c$: CellT | $c \in m \wedge$ belongToCell*(b, c)*: $c\}$=1*)*:$b\}$

C.7.4.4.12 boundaryVertices()

- **Description**
  Returns a set of boundary vertices of the mesh.
- **Output**
  $\{v$: `VertexT` $\mid v \in$ boundaryEdges*(): v*$\}$

C.7.4.5 Local Functions

C.7.4.6 Local Data Types

None

C.7.4.7 Local Constants

None

C.7.4.8 Considerations

None

C.8 MIS of Service Module

C.8.1 Module Name: Service

C.8.2 Uses

C.8.2.1 Imported Constants

None

C.8.2.2 Imported Data Types

Uses *Vertex Module* **Imports** `VertexT`
Uses *Edge Module* **Imports** `EdgeT`
Uses *Cell Module* **Imports** `CellT`
Uses *Mesh Module* **Imports** `MeshT`

C.8.2.3 Imported Access Programs

Uses *Mesh Module* **Imports** *onEdge(), inside(), vertices(), edges(), boundaryEdges(), boundaryVertices()*

C.8.3 Interface Syntax

C.8.3.1 Exported constants

None

C.8.3.2 Exported Data Types

`InstructionT` := {`REFINE, COARSEN, NOCHANGE`}
`CellInstructionT`:= tuple of (*cell*: `CellT`, *instr*: `InstructionT`)
`RCinstructionT` := tuple of
(*rORc*: `InstructionT`, *cInstru*: set of `CellInstructionT`)

C.8.3.3 Exported Access Programs

The exported access programs for the services module are listed in Table C.6.

Routine Name	Input	Output	Exceptions
isValidMesh	`MeshT`	$\mathbb{B}$	
coveringUp	`MeshT` × `MeshT`	$\mathbb{B}$	

Table C.6: Exported Access Programs of the Services Module

C.8.4 Interface Semantics

C.8.4.1 State Variables

None

C.8.4.2 Invariant

None

C.8.4.3 Assumptions

None

C.8.4.4 Access Program Semantics

C.8.4.4.1 isValidMesh(*m*: `MeshT`)

- **Description**
  Returns true if cells of the mesh are bounded, conformal, and non overlapping.

- **Output**
  Bounded(m) $\wedge$ Conformal(m) $\wedge$ NoInteriorIntersect(m)

C.8.4.4.2 coveringUp($m1$:MeshT, $m2$: MeshT)

- **Description**
  Returns false if any boundary vertex of one mesh is not on a boundary edge of another mesh. Otherwise, return true.

- **Output**
  $\forall v1, v2$: VertexT |
  $v1 \in$ boundaryVertice($m1$) $\wedge$ $v2 \in$ boundaryVertices($m2$) :
  ($\exists\, b1, b2$: EdgeT | $b1 \in$ boundaryEdges($m1$) $\wedge$ $b2 \in$ boundaryEdges($m2$):
  (onEdge($v1, b2$) $\vee$ $v1 \in b2$) $\wedge$ (onEdge($v2, b1$) $\vee$ $v2 \in b1$))

C.8.4.5 Local Functions

- ValidEdge: EdgeT $\rightarrow \mathbb{B}$
  ValidEdge(e: EdgeT) $\equiv \#e = 2$

- Area: CellT $\rightarrow \mathbb{R}$
  Area(c: CellT) $\equiv \Sigma v1, v2, v3$: VertexT | $v1 \in c \wedge v2 \in c \wedge v3 \in c$
  $\wedge\, v1 \neq v2 \wedge v2 \neq v3 \wedge v3 \neq v1$:
  $\frac{1}{12} * |v1.x * v2.y - v2.x * v1.y +$
  $v2.x * v3.y - v3.x * v2.y +$
  $v1.x * v3.y - v3.x * v1.y|$

- ValidCell: CellT $\rightarrow \mathbb{B}$
  ValidCell(c: CellT) $\equiv \#c = 3 \wedge$ Area(c) ≥ 0

- Bounded: MeshT $\rightarrow \mathbb{B}$
  Bounded(m: MeshT) $\equiv \forall v$: VertexT | $v \in$ boundaryVertices(m):
  ($\#\{e$: EdgeT | $e \in$ boundaryEdge(m) $\wedge v \in e : e\} = 2$)

- Conformal: MeshT $\rightarrow \mathbb{B}$
  Conformal(m: MeshT) $\equiv \forall c1, c2$: CellT | $c1 \in m \wedge c2 \in m \wedge c1 \neq c2$:
  ($\exists e$: EdgeT | $e \in$ edges(m)) : ($\exists v$: VertexT | $v \in$ vertices(m) :
  ($c1 \cap c2 = e \vee c1 \cap c2 = v \vee c1 \cap c2 = \emptyset$) $\wedge$ ($\neg$ onEdge(v, e))))

- NoInteriorIntersect: `MeshT` $\rightarrow \mathbb{B}$
  NoInteriorIntersect(m: `MeshT`) $\equiv \forall c1, c2$: `CellT` |
  $c1 \in m \wedge c2 \in m \wedge c1 \neq c2 : (\forall v$: `VertexT` | inside($v$, $c1$) : $\neg$ inside(v, $c2$))

C.8.4.6 Local Data Types

None

C.8.4.7 Local Constants

None

C.8.4.8 Considerations

None

C.9 MIS of Input Format Module

C.9.1 Module Name: Input Format

C.9.2 Uses

C.9.2.1 Imported Constants

None

C.9.2.2 Imported Data Types

Uses *Embedding Application* **Imports** `InputFormatT`

C.9.2.3 Imported Access Programs

None

C.9.3 Interface Syntax

C.9.3.1 Exported constants

None

C.9.3.2 Exported Data Types

None

C.9.3.3 Exported Access Programs

The exported access programs for the input format module are listed in Table C.7.

Routine Name	Input	Output	Exceptions
convertInput	`InputFormatT`	`MeshT`	

Table C.7: Exported Access Programs of the Input Format Module

C.9.4 Interface Semantics

C.9.4.1 State Variables

None

C.9.4.2 Invariant

None

C.9.4.3 Assumptions

None

C.9.4.4 Access Program Semantics

C.9.4.4.1 convertInput(m: `InputFormatT`)

- **Output**
  m' such that
  m' is of type `MeshT` and m and m' are equivalent.

C.9.4.5 Local Functions

None

C.9.4.6 Local Data Types

None

C.9.4.7 Local Constants

None

C.9.4.8 Considerations

- Semantics of access programs in this module heavily depend on the format of the input mesh. At this stage, this information is missing. Unknown data types `InputFormatT` is used to represent the data structures of input mesh. English is used to describe the semantics.

C.10 MIS of Output Format Module

C.10.1 Module Name: Output Format

C.10.2 Uses

C.10.2.1 Imported Constants

None

C.10.2.2 Imported Data Types

Uses *Embedding Application* **Imports** `OutputFormatT`

C.10.2.3 Imported Access Programs

None

C.10.3 Interface Syntax

C.10.3.1 Exported constants

None

C.10.3.2 Exported Data Types

None

C.10.3.3 Exported Access Programs

The exported access programs for the output format module are listed in Table C.8.

Routine Name	Input	Output	Exceptions
convertOutput	`MeshT`	`OutputFormatT`	

Table C.8: Exported Access Programs of the Output Format Module

C.10.4 Interface Semantics

C.10.4.1 State Variables

None

C.10.4.2 Invariant

None

C.10.4.3 Assumptions

None

C.10.4.4 Access Program Semantics

C.10.4.4.1 convertOutput(m: `MeshT`)

- **Output**
  m' such that
  m' is of type `OutputFormatT` and m and m' are equivalent.

C.10.4.5 Local Functions

•

None

C.10.4.6 Local Data Types

None

C.10.4.7 Local Constants

None

C.10.4.8 Considerations

- Semantics of access programs in this module heavily depend on the format of the requirements of the output. At this stage, this information is missing. Unknown data type `OutputFormatT` are used to represent the data structures of input and output. English is used to describe the semantics.

C.11 MIS of Refining Module

C.11.1 Module Name: Refining

C.11.2 Uses

C.11.2.1 Imported Constants

None

C.11.2.2 Imported Data Types

Uses *Mesh Module* **Imports** `MeshT`
Uses *Service Module* **Imports**
`InstructionT, CellInstructionT, RCinstructionT`

C.11.2.3 Imported Access Programs

Uses *Service Module* **Imports** *isValidMesh(), coveringUp()*

C.11.3 Interface Syntax

C.11.3.1 Exported constants

None

C.11.3.2 Exported Data Types

None

C.11.3.3 Exported Access Programs

The exported access programs for the refining module are listed in Table C.9.

Routine Name	Input	Output	Exceptions
refining	`MeshT, RCinstructionT`	`MeshT`	

Table C.9: Exported Access Programs of the Refining Module

C.11.4 Interface Semantics

C.11.4.1 State Variables

None

C.11.4.2 Invariant

None

C.11.4.3 Assumptions

isValidMesh(m) and $i.rORc =$ `REFINE`
for input m: `MeshT` and i: `RCinstructionT`

C.11.4.4 Access Program Semantics

C.11.4.4.1 refining(m: `MeshT`, i: `RCinstructionT`)

- **Output**
  m'
  such that
  ValidMesh(m) $\wedge$ ValidMesh(m') $\wedge$ CoveringUp(m', m) $\wedge \#m' \geq \#m$

C.11.4.5 Local Functions

None

C.11.4.6 Local Data Types

None

C.11.4.7 Local Constants

None

C.11.4.8 Considerations

None

C.12 MIS of Coarsening Module

C.12.1 Module Name: Coarsening

C.12.2 Uses

C.12.2.1 Imported Constants

None

C.12.2.2 Imported Data Types

Uses *Mesh Module* **Imports** `MeshT`
Uses *Service Module* **Imports**
`InstructionT, CellInstructionT, RCinstructionT`

C.12.2.3 Imported Access Programs

Uses *Service Module* **Imports** *isValidMesh(), coveringUp()*

C.12.3 Interface Syntax

C.12.3.1 Exported constants

None

C.12.3.2 Exported Data Types

None

C.12.3.3 Exported Access Programs

The exported access programs for the coarsening module are listed in Table C.10.

Routine Name	Input	Output	Exceptions
coarsening	`MeshT, RCinstructionT`	`MeshT`	

Table C.10: Exported Access Programs of the Coarsening Module

C.12.4 Interface Semantics

C.12.4.1 State Variables

None

C.12.4.2 Invariant

None

C.12.4.3 Assumptions

isValidMesh(m) and $i.rORc =$ `COARSEN`
for input m: `MeshT` and i: `RCinstructionT`

C.12.4.4 Access Program Semantics

C.12.4.4.1 coarsening(m: `MeshT`)

- **Output**
  m'
  such that
  ValidMesh(m) $\wedge$ ValidMesh(m') $\wedge$ CoveringUp(m', m) $\wedge \#m' \leq \#m$

C.12.4.5 Local Functions

None

C.12.4.6 Local Data Types

None

C.12.4.7 Local Constants

None

C.12.4.8 Considerations

None

Appendix D

The Summary of Validation Testing Report for a Parallel Mesh Generation Toolbox

D.1 Introduction

This section gives an overview of the Testing Summary for a Parallel Mesh Generation Toolbox (PMGT). First, the purpose of the document is provided. Second, the scope of the testing is identified. Third, the organization of the document is summarized.

D.1.1 Purpose of the Document

This document specifies validation tests for a PMGT. The results of the tests and analysis are also provided. The intended audience is testers who are going to test the system and developers who are going to maintain the software. Note that test document is dynamic in the sense that it should be updated when the development of the system proceeds.

D.1.2 Scope of the Testing

In general, the purpose of testing is to help produce quality software. Due to limits on the time available for testing, the scope of the testing of PMGT is restricted to test the most important test factors. Like other scientific computing software, correctness and efficiency are considered to be the two most important test factors for PMGT. For efficiency testing, the focus is on execution time rather than on storage.

D.1.3 Organization of the Document

Section D.1 (this section) is an introduction to the report. Section D.2 shows what is going to be tested and the coverage of the testing, with respect to the software requirements and the software design. Section D.3 gives the result of the testing and the analysis.

D.2 Testing PMGT

Test cases are listed in Section D.2.1. The detailed information for these test cases can be found in Section D.3. The traceability matrix in Section D.2.2 shows the association between test cases and the functional and nonfunctional requirements that are specified in the Software Requirements Specification (SRS) document. Similarly, a traceability matrix for test cases and the leaf modules as introduced in the Module Guide (MG) as shown in Section D.2.3. Tracking these relations is useful for developing and maintaining the software.

D.2.1 Test Cases

The correctness validation test is designed for verifying the functional requirements RefiningMesh (F1), CoarseningMesh (F2), ElmShape (F5), and Conformal (F7). Other requirements for correctness are trivial and are satisfied obviously. For example, since the vertices are stored in an array, the OutVertexOrder (F15) requirement is met by outputting the vertices in the order as the order of them in the array. The tests are against above requirements are automated. The automated validation tests requirements (ACVTRs) are listed in Section D.2.1.1. Since the output mesh also can also be displayed on screen, it can be checked manually. The visual correctness validation tests requirements (VCVTRs) are listed in Section D.2.1.2. The test cases are in Section D.2.1.3.

D.2.1.1 Automated Correctness Validation Tests Requirements

A list of ACVTRs follows. All test cases should pass these tests. Some test cases relate to data definitions defined in the SRS. In these cases the related data definition defined is shown as Dx, where x is the number of the associated data definition given in the SRS.

- The area of each element is greater than zero (referring to D5).
- The boundary of the mesh is closed. (referring to D15).
- The mesh is conformal (referring to D16).
- The intersection of any two elements is empty (referring to D17).
- The input mesh and output mesh *CoveringUp* each other (referring to D19).
- The length of each edge is greater than zero. (This is required by the definition of a mesh, which is defined in the SRS.)
- The vertices of each element are listed in a counterclockwise order. (The counterclockwise order of the vertices for each element is not necessary for implementing PMGT. However, it is adopted by most meshing and FEA software. PMGT uses this convention.)
- The output mesh conforms to the Euler Equation. (This requirement is not documented in the SRS. However, any mesh should implicitly satisfy the equation $nc + nv - ne = 1$, where nc is the number of cells, nv is the number of vertices, and ne is the number of edges.)

D.2.1.2 Visual Correctness Validation Tests Requirements

The output meshes should also be visually checked to ensure that the following VCVTRs are met.

- No vertex is outside of the input domain.
- No vertex is inside of a cell.
- No dangling points or edges are present.
- All cells are connected.
- The mesh is conformal.

Some of the VCVTRs overlap with the ACVTRs. This redundancy provides increased confidence in case one testing method fails to catch an error.

D.2.1.3 Test Cases

The test cases developed involve testing meshes against the above requirements. In each test case, except the last one, the input mesh is refined and then coarsened. Two algorithms for refining are used. One algorithm is called *Split*. It splits one cell into three by adding a point in the centroid of the triangle and connecting the added point to the three original vertices. The other algorithm is simply call *Refine*. It refines the original mesh by longest edge bisection.

The name of each test case includes three parts. For example, test case $AxxCBN$ means that the test uses Axx algorithm for refining, where Axx equals *Split* or $Refine$. The letter C indicates that coarsening is performed. If the C is missing, the input mesh is not coarsened. B is the number of refinements before coarsening. If B is S, the mesh is refined once and then coarsen once. If the B is M, the mesh is refined multiple time before coarsening. N is a number. If the N is omitted, it means only one of this kind of test performed. Otherwise the same test procedures is used several times on different input meshes. The reason for using the same procedure is that the topology of the output meshes may differ for different input meshes.

- Test Case *SplitCS* (TC1): This test case tests the correctness of PMGT. The input mesh is shown in Figure D.3. The refining and coarsening criterion is that the cells intersected with the vertical line, $x = 0.6$, are Split once, then the cells of the new mesh that intersect with the vertical line are coarsened once. When the splitting and coarsening is done, the

vertical line is moved to the right one unit ($x = x + 1.0$), and another Splitting and coarsening is performed. This procedure is repeated until no cells intersect with the vertical line.

- Test Case *RefineCS1* (TC2): This test case tests the correctness of PMGT. The input mesh is the same as TC1, which is shown in Figure D.3. There is a vertical line at $x = 0.6$. The refining and coarsening criterion is that the cells that intersect with the vertical line are refined once, then the cells of the new mesh that intersect with the vertical line are coarsened once. When the refining and coarsening are done, the vertical line is moved to the right one unit, and another refining and coarsening is performed. This procedure is repeated until no cells intersect with the vertical line.

- Test Case *RefineCS2* (TC3): This test case tests the correctness of PMGT. The refining and coarsening criterion, vertical line function, and the test procedure are the same as test case TC2. However, the input mesh is different. The input mesh is showed in Figure D.4.

- Test Case *RefineCM* (TC4): This test case tests the correctness of PMGT. The input mesh is shown in Figure D.5. There is a vertical line at $x = 0.5$. The refining and coarsening criterion is the size of the cells. The size of the cell is measured by the length of the longest edge of the cell. The cells that intersect with the vertical line are refined until the criterion is met. When the refining is done, the vertical line is moved to the right 0.6 unit ($x = x + 0.6$), and another refinement is performed. After five refinements are done, the cells to be left of the vertical line by up to 2 units are coarsened, until the coarsening criterion is met. The refining and coarsening are stopped when the vertical line moves to a position outside of the domain.

- Test Case *RefineM* (TC5): This test case tests the correctness of PMGT. The input mesh is shown in Figure D.6. There is an arc with radius of 0.7 unit going through the mesh. Cells that intersect with the arc are refined until the required number of refinements has been reached.

- Test Case *Split* (TC6): This test case tests both the correctness and speed of PMGT. The input mesh is shown in Figure D.7. This test simply splits all cells of the mesh 4 times. It is done in both the serial version and the parallel version with different number of processors. The execution time of setting the cells to be refined and splitting the cells is measured.

D.2.2 Traceability Matrix for SRS

In the traceability matrix for software requirements, if a test case tests the functionality of a software requirement, there will be a check mark on the cell for the corresponding test case. In each row of the traceability matrix for software requirements (Table D.1), if the requirement in that row defines the correctness or the speed of the software, one or more cells in this row are checked. Otherwise, all cells in the row are empty. Table D.1 shows that the test cases developed in Section D.2.1 assist with validating the correctness and speed of the software. The detailed information for each functional and nonfunctional requirements can be found in the SRS document. The names of the requirements and their corresponding numbers are listed below for convenience.

F1: RefiningMesh

F2: CoarseningMesh

F3: RefiningOrCoarsening

F4: MeshType

F5: ElmShape

F6: DomainDimension

F7: Conformal

F8: InputDefinition

F9: RCInstruction

F10: OutputStorage

F11: VertexUniqueID

F12: ElmUniqueID

F13: ElmTopology

F14: OutElmOrder

F15: OutVertexOrder

F16: Help

N1: Performance

N2: Precision

N3: Exception

N4: Portability

N5: LookAndFeel

N6: Usability

N7: Maintainability

	TC1	TC2	TC3	TC4	TC5	TC6
F1	✓	✓	✓	✓	✓	✓
F2	✓	✓	✓	✓		
F3	✓	✓	✓	✓	✓	✓
F4	✓	✓	✓	✓	✓	
F5	✓	✓	✓	✓	✓	
F6	✓	✓	✓	✓	✓	
F7	✓	✓	✓	✓	✓	
F8	✓	✓	✓	✓	✓	✓
F9	✓	✓	✓	✓	✓	✓
F10	✓	✓	✓	✓	✓	
F11	✓	✓	✓	✓	✓	
F12	✓	✓	✓	✓	✓	
F13	✓	✓	✓	✓	✓	
F14	✓	✓	✓	✓	✓	
F15	✓	✓	✓	✓	✓	
F16						
N1						✓
N2						
N3						
N4						
N5						
N6						
N7						

Table D.1: Traceability Matrix: Test Cases and Requirements

D.2.3 Traceability Matrix for MG

Similar to Section D.2.2, the traceability matrix for modules (Table D.2) shows that the test cases validate the modules that are associated with correctness and speed. The names of modules appear in Table D.2 are listed below. The detailed information for each module can be found in the MG document.

M1: Virtual Memory Module

M2: File Read/Write Module

M3: Keyboard Input Module

M4: Screen Output Module

M5: Input Format Module

M6: Output Format Module

M7: Service Module

M8: Vertex Module

M9: Edge Module

M10: Cell Module

M11: Mesh Module

M12: Refining Module

M13: Coarsening Module

D.3 Results and Analysis

The results of the test cases defined in Section D.2.1.3 are listed in Section D.3.1. The analysis, including charts that compare the execution time of the parallel version to the serial version are provided in Section D.3.2.

	TC1	TC2	TC3	TC4	TC5	TC6
M1	✓	✓	✓	✓	✓	✓
M2	✓	✓	✓	✓	✓	✓
M3	✓	✓	✓	✓	✓	✓
M4	✓	✓	✓	✓	✓	✓
M5	✓	✓	✓	✓	✓	✓
M6	✓	✓	✓	✓	✓	✓
M7	✓	✓	✓	✓	✓	
M8	✓	✓	✓	✓	✓	✓
M9	✓	✓	✓	✓	✓	✓
M10	✓	✓	✓	✓	✓	✓
M11	✓	✓	✓	✓	✓	✓
M12	✓	✓	✓	✓	✓	✓
M13	✓	✓	✓	✓		

Table D.2: Traceability Matrix: Test Cases and Modules

D.3.1 Testing Results

The following tables list the testing results of each test case. The field *Test Case Number* and *Test Case Name* list the number and the name of each test case. The *Input field* gives the number of the figure that is the input for that test case, or a description of the input mesh. The *Expected Output* describes the requirements of the output mesh. The *Actual Output* gives the result of the test. The *Selected Output Mesh* field should give the output meshes. However, there are too many intermediate mesh to display, and displaying only the final mesh is too simple to illustrate the feature of the test case. Selected intermediate meshes and final mesh are included in the *Actual Output* field. The *Result* field indicates whether the test is passed or failed.

Test Case Number **Test Case Name**	TC1 SplitCS
Input	Figure D.3
Expected Output	ACVTRs and VCVTRs listed in Section D.2 are met
Actual Output	Summary of the correctness test: 15 tests are performed. 15 tests succeed. 0 tests fail.
Selected Output Mesh	Figure D.8, D.9, D.10
Result	Passed

Test Case Number **Test Case Name**	TC2 RefineCS1
Input	Figure D.3
Expected Output	ACVTRs and VCVTRs listed in Section D.2 are met
Actual Output	Summary of the correctness test: 15 tests are performed. 15 tests succeed. 0 tests fail.
Selected Output Mesh	Figure D.11, D.12, D.13
Result	Passed

Test Case Number **Test Case Name**	TC3 RefineCS2
Input	Figure D.4
Expected Output	ACVTRs and VCVTRs listed in Section D.2 are met
Actual Output	Summary of the correctness test: 15 tests are performed. 15 tests succeed. 0 tests fail.
Selected Output Mesh	Figure D.14, D.15, D.16
Result	Passed

Test Case Number	TC4
Test Case Name	RefineCM
Input	Figure D.5
Expected Output	ACVTRs and VCVTRs listed in Section D.2 are met
Actual Output	Summary of the correctness test: 15 tests are performed. 15 tests succeed. 0 tests fail.
Selected Output Mesh	Figure D.17, D.18, D.19, D.20
Result	Passed

Test Case Number	TC5
Test Case Name	RefineM
Input	Figure D.6
Expected Output	ACVTRs and VCVTRs listed in Section D.2 are met
Actual Output	Summary of the correctness test: 15 tests are performed. 15 tests succeed. 0 tests fail.
Selected Output Mesh	Figure D.21, D.22, D.23
Result	Passed

Test Case Number	TC6
Test Case Name	SplitM
Input	Figure D.6
Expected Output	ACVTRs and VCVTRs listed in Section D.2 are met Execution time increases as the number of cells increases. Execution time decreases as the number of processors increases.
Actual Output	Execution time as indicated in Figure D.1
Selected Output Mesh	The mesh is too dense to be shown.
Result	Passed

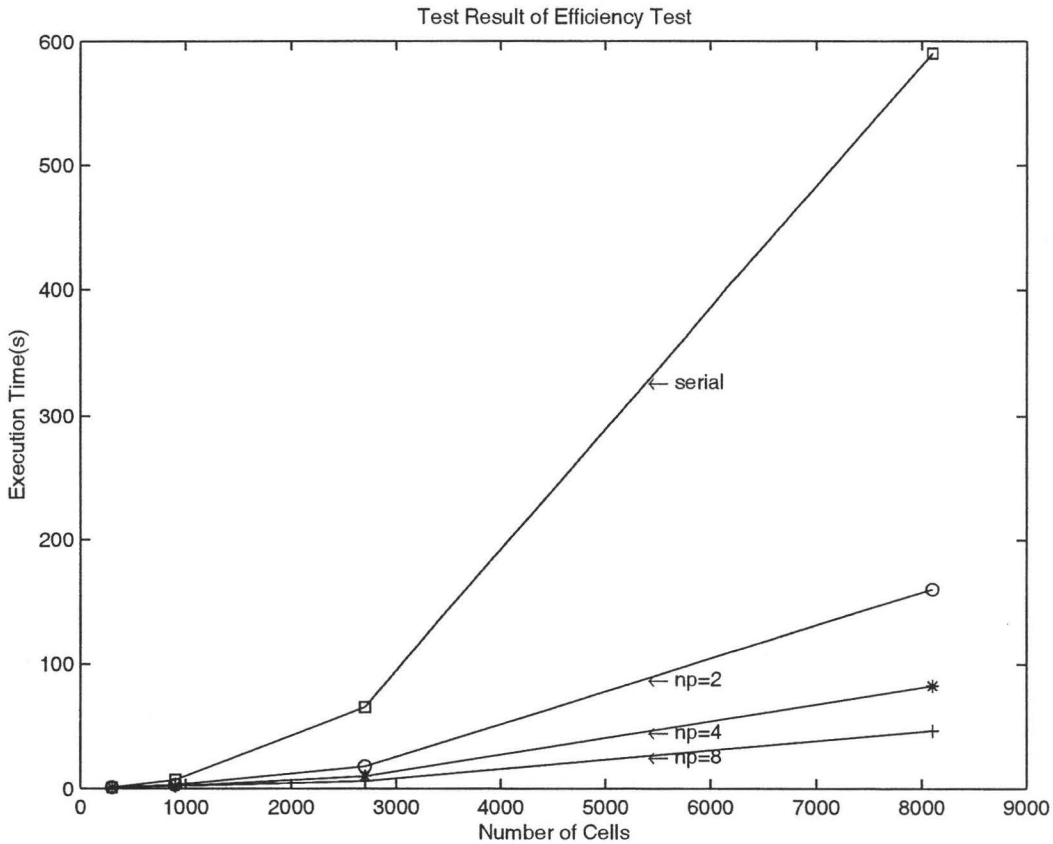

Figure D.1: Output of TC6

D.3.2 Analysis

All of the test cases conform to the ACVTRs and VCVTRs listed in Section D.2. The test result of TC6 show that when the number of cells increased, the execution time increased, and when the number of processors increased, the execution time decreased. That is, this test is passed. Figure D.2 show the speedup when using different numbers of processors. The speedup is defined as

$$Speedup(n) = \frac{T_1}{T_n}$$

Where T_1 is the execution time of the serial version, and T_n is the execution time of the parallel version with n processors. In general, $Speedup(n) < n$. However, for PMGT, when the number of cells is greater than 2700, $Speedup(n) > n$, which is a super linear speedup. Since the algorithms used for the serial version and the parallel version are the same, the super linear

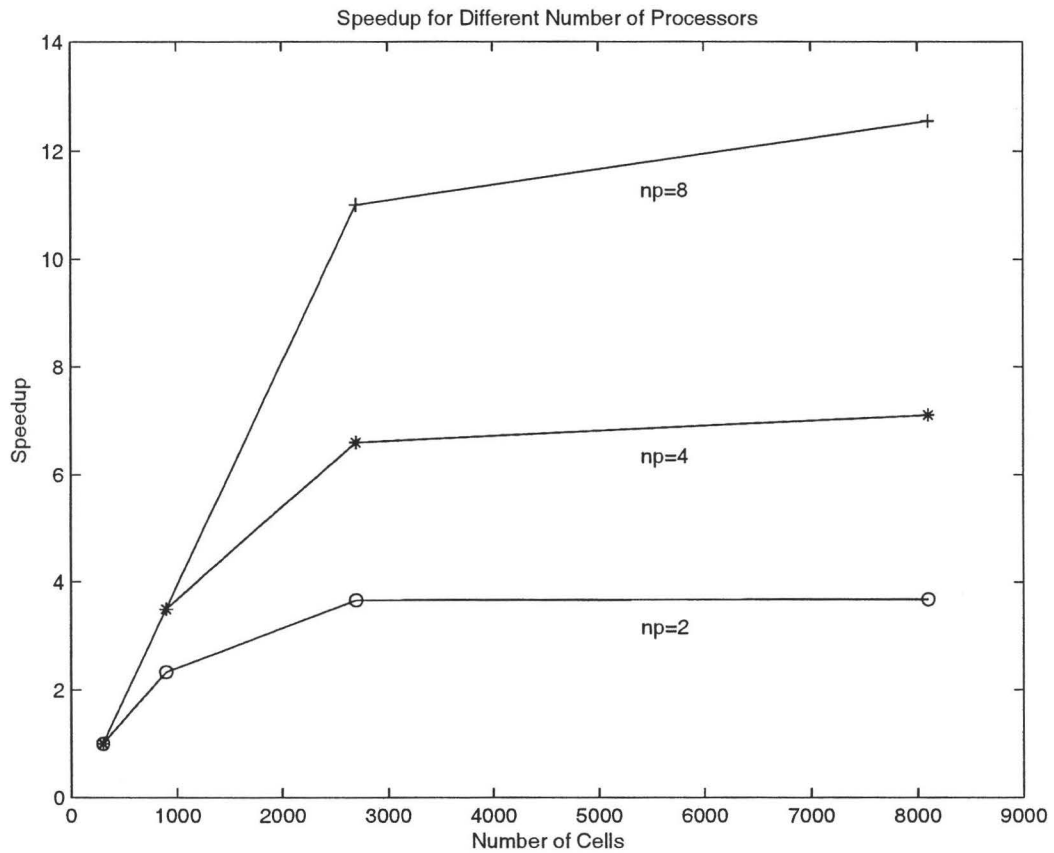

Figure D.2: Speedup for Different Numbers of Processors

speedup is probably due to the cache effect. That is, when the numbers of processors increases, the size of the accumulated caches from different processors also increases. With the larger accumulated cache size, more, or even all, core data set can fit into the caches and the memory access time reduces dramatically. This may explain the extra speedup in additional to the speedup due to parallel computation.

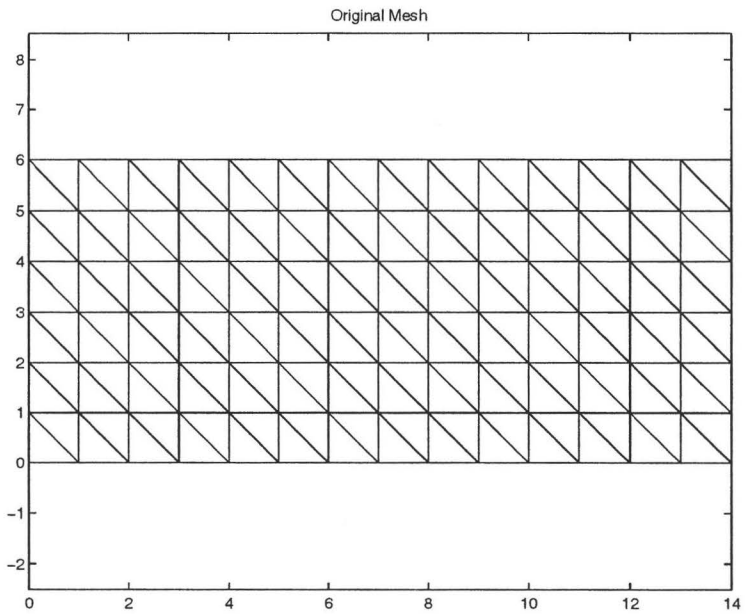

Figure D.3: Input 1

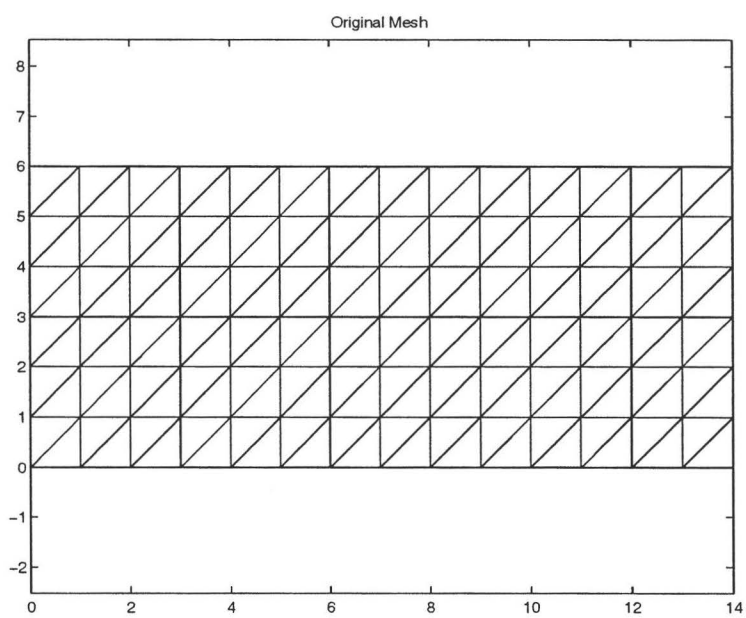

Figure D.4: Input 2

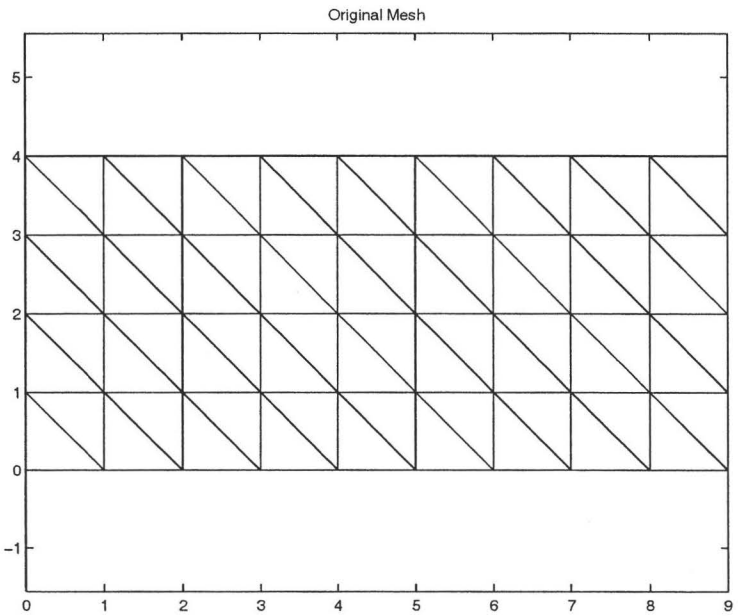

Figure D.5: Input 3

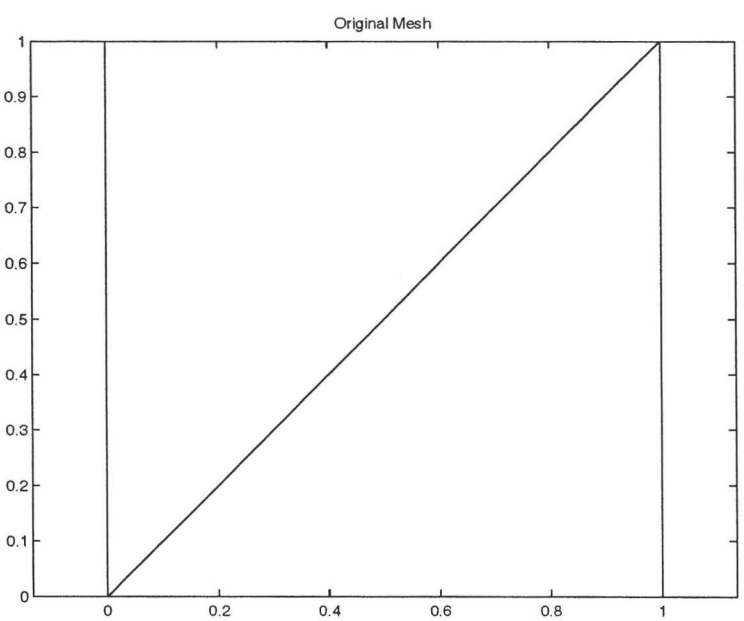

Figure D.6: Input 4

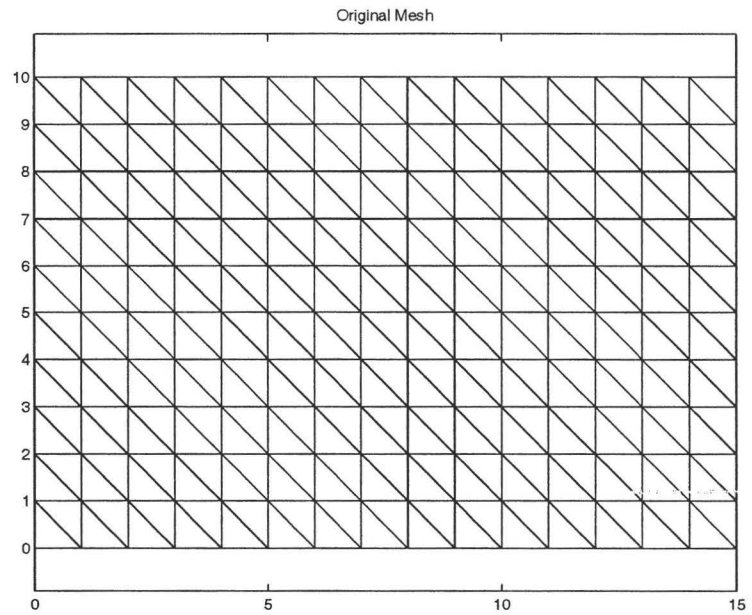

Figure D.7: Input 5

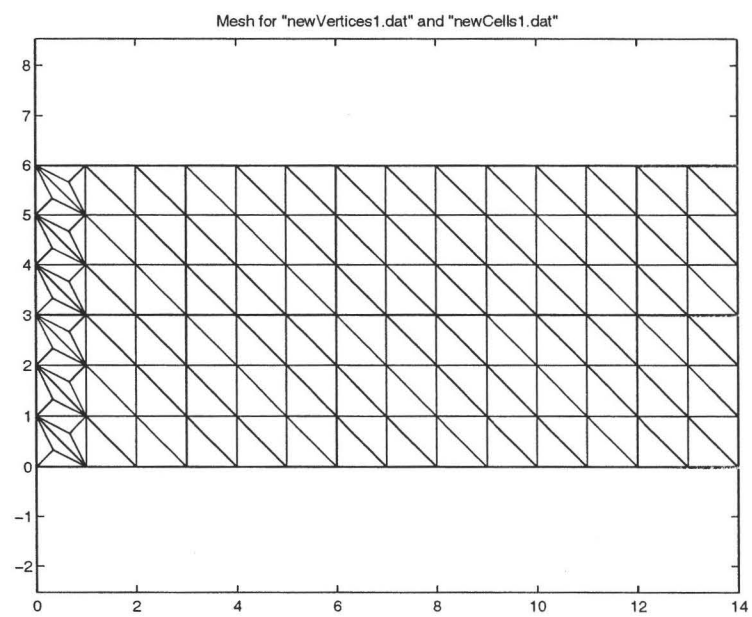

Figure D.8: Output 1 of TC1

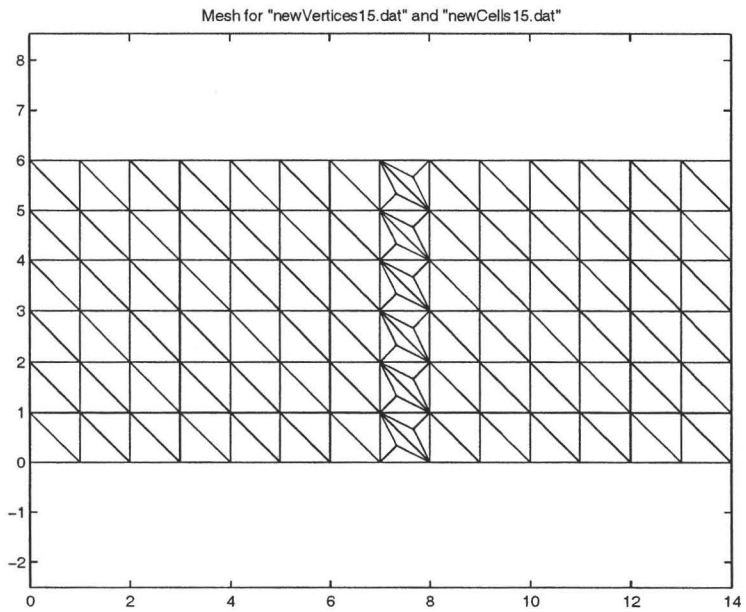

Figure D.9: Output 2 of TC1

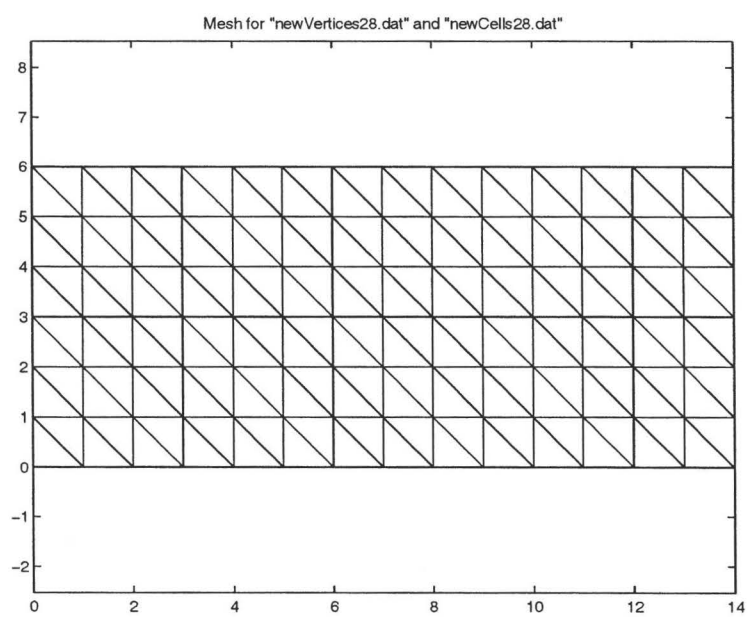

Figure D.10: Output 3 of TC1

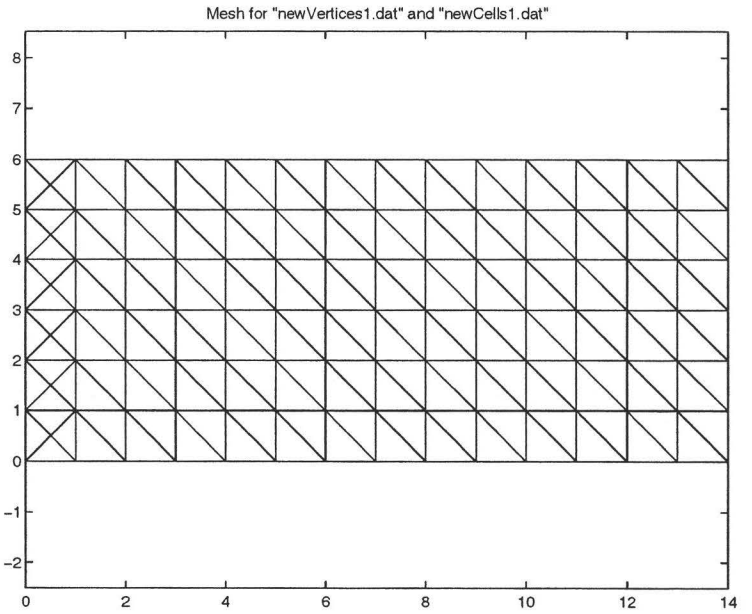

Figure D.11: Output 1 of TC2

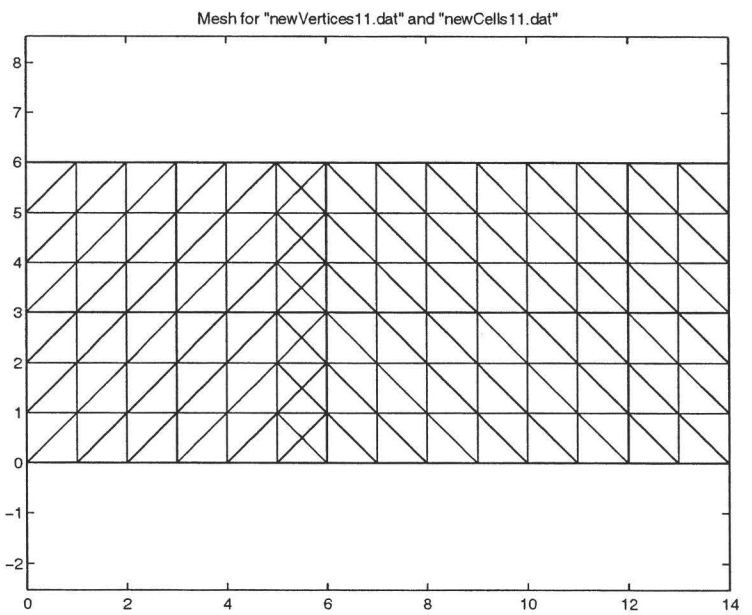

Figure D.12: Output 2 of TC2

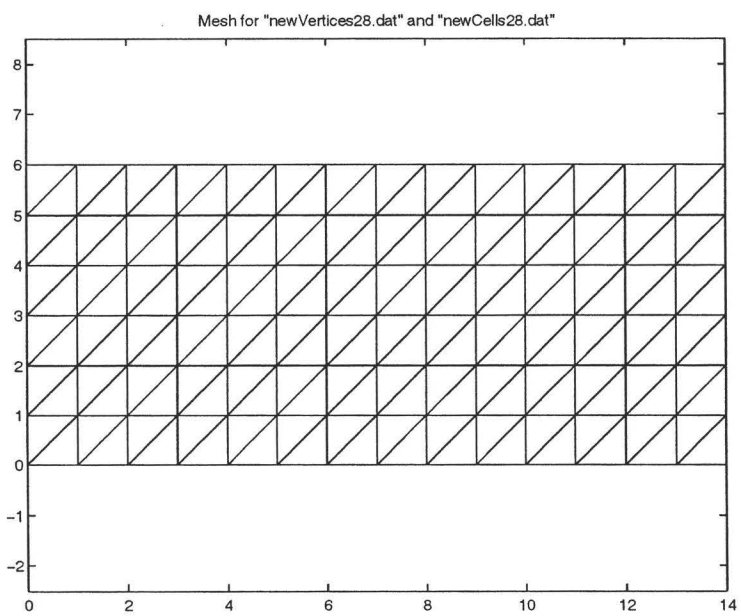

Figure D.13: Output 3 of TC2

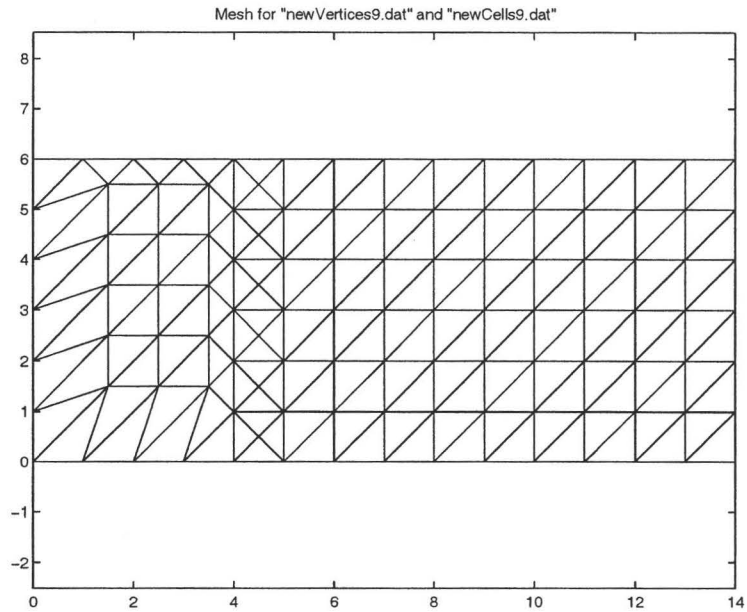

Figure D.14: Output 1 of TC3

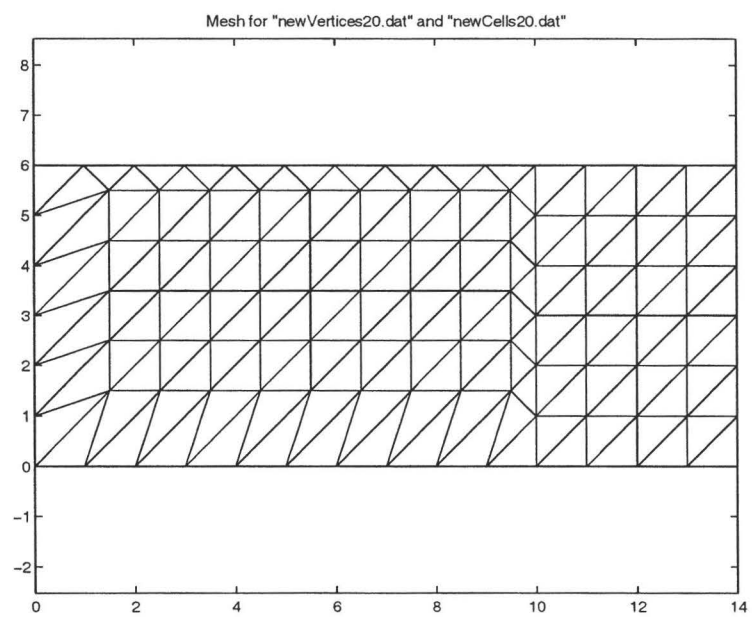

Figure D.15: Output 2 of TC3

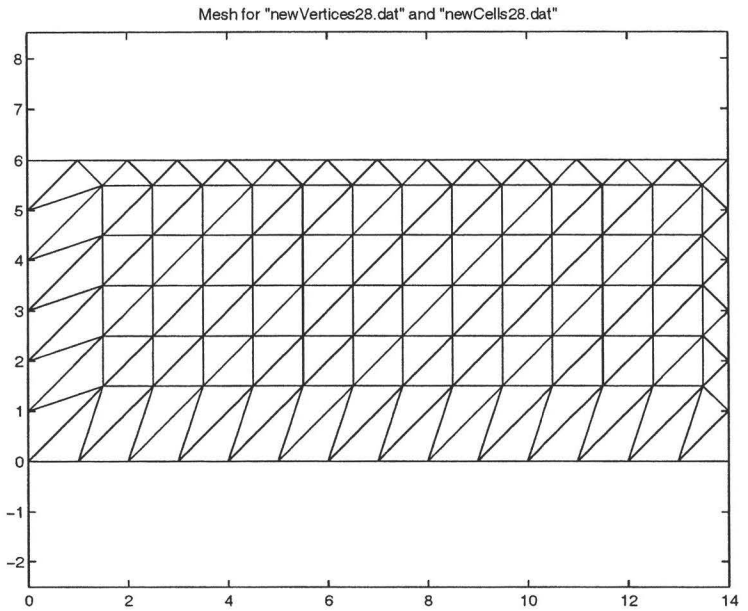

Figure D.16: Output 3 of TC3

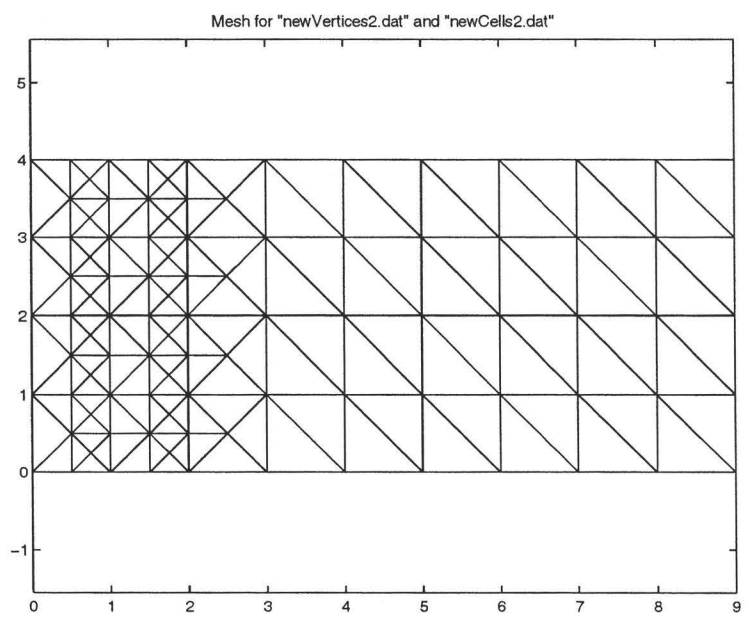

Figure D.17: Output 1 of TC4

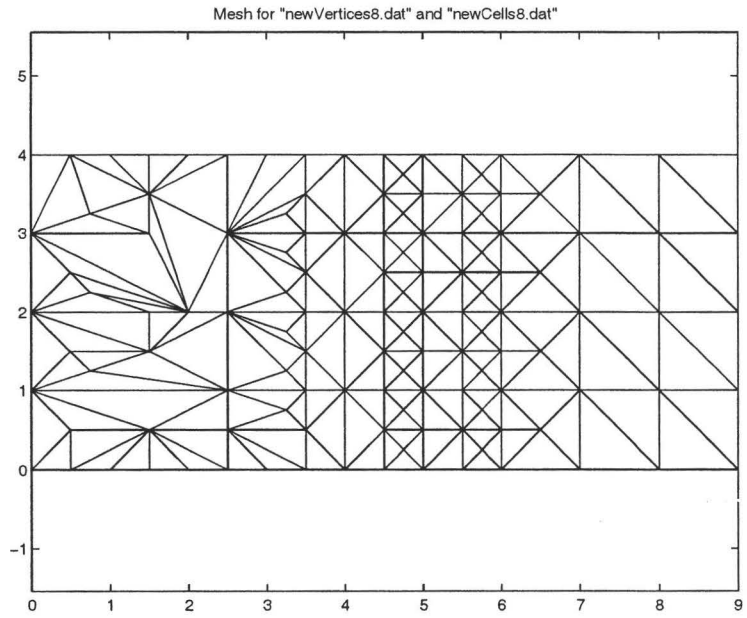

Figure D.18: Output 2 of TC4

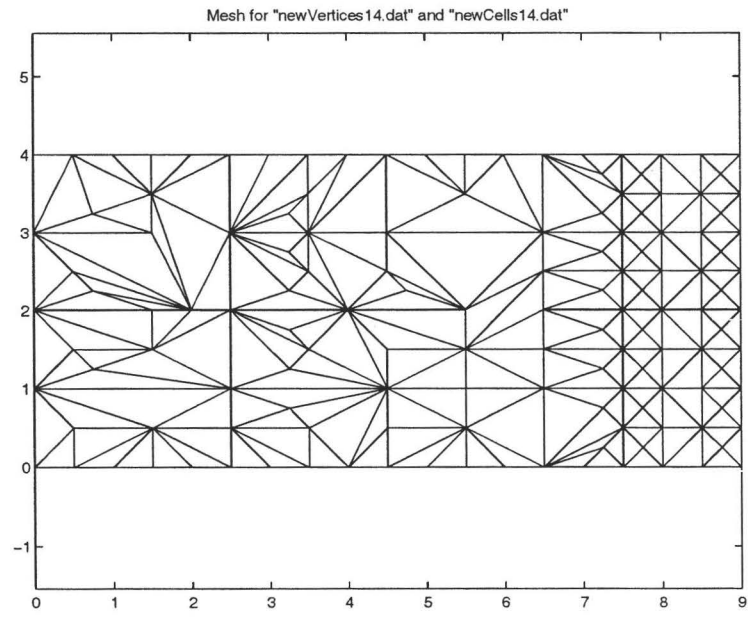

Figure D.19: Output 3 of TC4

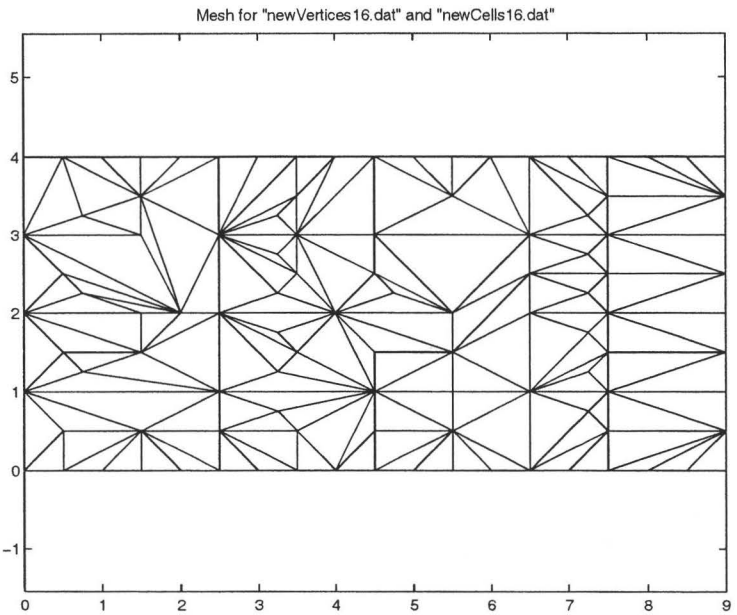

Figure D.20: Output 4 of TC4

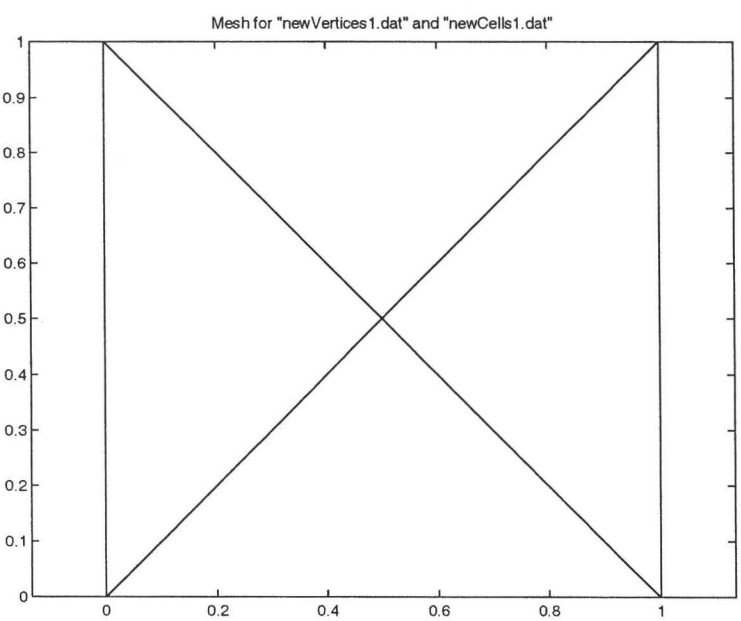

Figure D.21: Output 1 of TC5

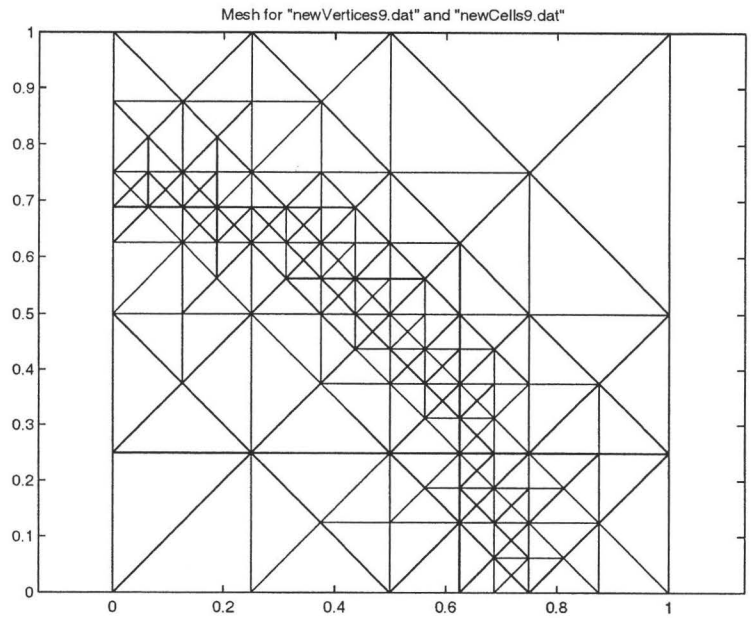

Figure D.22: Output 2 of TC5

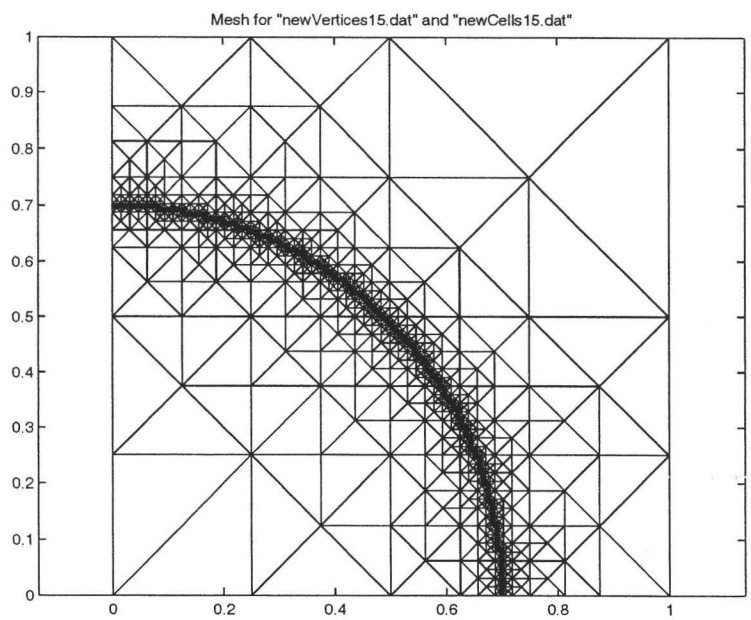

Figure D.23: Output 3 of TC5