A PROGRAM FAMILY APPROACH TO DEVELOPING MESH GENERATORS

A PROGRAM FAMILY APPROACH TO DEVELOPING MESH GENERATORS

By
FANG CAO, B.Sc

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Applied Science

McMaster University

MASTER OF APPLIED SCIENCE (2006 April) McMaster University

(Software Engineering) Hamilton, Ontario

TITLE: A Program Family Approach to Developing Mesh Generators

AUTHOR: Fang Cao, B.Sc (University of Ottawa)

SUPERVISORS: Dr. Spencer Smith

NUMBER OF PAGES: vii, 211

Acknowledgement

First of all, I would like to thank my supervisor, Dr. Spencer Smith, for his suggestions, guidance, and motivation during my pursuit of the Master degree. Without his encouragement and support, this work would not have come to an existence. For that I'm eternally grateful for his help.

I also want to express my sincere gratitude to my family, especially my parents and my girlfriend Sandy. They have been such a blessing in my life and given me constant support and encouragement.

Abstract

This thesis presents a systematic approach for rapid development of a program family of special-purpose 2D structured mesh generators, where a mesh is a discretization of a geometric domain into small simple shapes, such as triangles or quadrilaterals. Mesh generators are commonly used to produce the input files for finite element and other numerical analysis programs that solve partial differential equations.

Despite the existence of many general-purpose mesh generators, there is a lack of research attention on the design of special-purpose generators suitable for small and specific meshing problems. The program family approach shown in this thesis supports reuse and code customization by identifying the commonalities and variabilities between mesh generators. The program family we developed accommodates variabilities in geometry, topology, material properties, boundary conditions, system parameters and output file format. Developing mesh generators using the program family approach results in the quality of special purpose mesh generators being improved in terms of usability, reusability and maintainability.

The contribution of this thesis centres on the design documentation and the system implementation. The complete documentation of our design includes a commonality analysis, requirements specification, and module guide. The contributions with respect to the implementation include the use of a domain-specific language (DSL) written in XML (Extensible Mark-up Language) to model the seed specification required to produce a mesh generator family member, as well as applying XSL (Extensible Stylesheet Language) to allow flexible customization of the output file(s).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Meshing can be defined as the process of decomposing a spatial domain into smaller and simpler sub-domains. Mesh generation techniques are widely employed in essentially all engineering fields as part of the process of modeling physical phenomena. For instance, a generated mesh serves as input to a finite element program, which can be used to solve partial differential equations for the purpose of analysis and design. Due to the potential complexity and size of the spatial domains of interest, a computerized approach is often required to produce a suitable mesh. The software used to automatically create the mesh is called a mesh generator.

Despite the excellent progress of research on the development of mesh generation algorithms, from a software development perspective there is still progress to be made. For example, options for formatting the output file(s) in many mesh generators are limited and only rarely does the user have the capability for customization of the output file formats. Another problem with the current development of mesh generators is the lack of sufficient software documentation, especially with respect to requirement and design specifications. In part due to this lack of documentation, many similar mesh generators have been developed with little reuse of existing designs and code. In software engineering, when there is a collection of similar programs like this, the programs can often be developed as a program family. If development as a program family can be shown to be feasible, then this will facilitate a systematic approach to rapidly developing mesh generators. Moreover, when a family-oriented approach is combined with software engineering methodologies for design and documentation, the quality of mesh generators can be improved not only with respect to reusability, but also with respect to usability, portability, and maintainability. The goal of this thesis is to show that a program family approach is feasible and that software engineering methodologies can be employed to allow rapid development of a high quality program family of mesh generators.

This chapter is intended to provide the reader with introductory material and background information for the later chapters. First, we present an overview of mesh terminologies. Second, we look at the current development of mesh generators and evaluate existing design strategies. Third, we briefly review methodologies for developing and generating program families. Fourth, we introduce the motivation, the goals, and the scope of our work, and finally we present an overview of the organization for the remainder of the thesis.

1.1 Background on Mesh Generators

This section provides background knowledge about meshes and mesh generation software. Section 1.1.1 introduces the geometric nature of meshes and defines the common terms employed in the field of mesh generation. Section 1.1.2 elaborates on the previous section and describes non-geometric mesh information, such as material properties and boundary conditions. This non-geometric information is needed when the mesh generator is used as a finite element pre-processor.

1.1.1 Mesh Terminology

Informally, a mesh is a discretization of a geometric domain into small simple shapes, such as line segments in 1D, triangles or quadrilaterals in 2D, and tetrahedral or hexahedra in 3D. Figure 1.1 is a hexahedral mesh used for numerical simulation of drop testing of a latch mechanism. (The volume mesh in Figure 1.1 is slightly changed from the original mesh available at http://cubit.sandia.gov/help-version9.1/chapter_2/gui_tutorial/images/hex_mesh.gif)

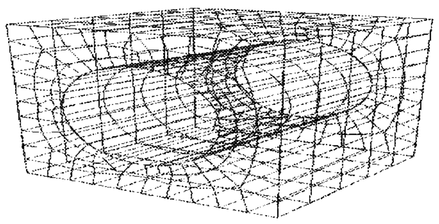


Figure 1.1: A Hexahedral Mesh

A formal definition of mesh can be found in [47] as follows:

Let Ω be a closed bounded domain in $\mathbb{R}$ or $\mathbb{R}^2$ or $\mathbb{R}^3$ and let K be a simple shape, such as a line segment in 1D, a triangle or a quadrilateral in 2D, or a tetrahedron or hexahedron in 3D. A mesh of Ω, denoted by Ω^*, has the following properties:

1. $\Omega \approx \bigcup(K|\ K \in \Omega^* : K)$, where $\bigcup$ 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 3, in Ω^* is nonempty.
4. The intersection of the interior of two elements is empty.

The above definition of a mesh is slightly modified from the original definition found in [18]. First of all, the "=" sign is changed to "$\approx$" because the discretization of the computational domain may not always exactly match the boundary of the domain. For example, if the boundary is curved, it cannot be matched with a mesh consisting of a finite number of straight edged triangular elements. Second, one-dimensional elements have been added to the definition.

Meshes are widely used in many application areas. For instance, in geography and cartography, meshes are employed to give precise representations of terrain data [4]. In computer graphics, most objects are first reduced to meshes before they are rendered to the screen. Meshes are also of critical importance for the finite element method, which will be the principle application of interest in our study. The finite element method is employed to numerically solve partial differential equations (P.D.Es) that arise in physical simulation [18]. The first step in a finite element analysis is the generation of a finite element mesh. The output of the mesh generation program becomes the input to a finite element program. In the finite element method, the quality of the mesh, in terms of the size, shape, and placement of the elements, is critical to the success of the finite element analysis. Advanced mesh optimization and refinement techniques such as smoothing [9, 36] and adaptive meshing [3] have been implemented as ways to produce higher quality meshes.

A mesh generator used as finite element pre-processor produces a mesh that contains three types of information: geometry, topology(connectivity), and physical information. This section focuses on the geometric and topological information, with the discussion of physical information postponed until the next section.

Figure 1.2 shows the following geometric entities of a mesh: domain, boundary, vertex, edge, and element. A domain, or computational domain, is a spatial area or volume that is to be discretized. The boundary of a mesh encloses the domain. A vertex is a point in space that defines the shape of a cell. In 1D, the vertices are end-points of elements. In 2D and 3D elements, the vertices are the locations where edges of elements intersect. The location of the vertices in a mesh are given by their coordinates. An edge of a mesh connects two vertices. In mesh generation, the domain is discretized into smaller, and simpler shapes called elements or cells. The domain of each element is delimited by the set of its vertices.

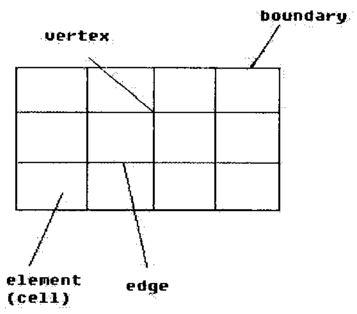


Figure 1.2: Mesh Entities in a Computational Domain

The connectivity of a mesh provides the topological pattern reflected by the mesh and its elements. There are two types of topological information: one for the mesh, and one for a mesh element. The connectivity of a mesh element is called cell connectivity and is defined as " the definition of the connections between the vertices at the element level" [18], whereas the connectivity of the mesh is given by the set of connectivities of its constituent elements. In the output file(s) produced by a mesh generator, the cell connectivity information lists the neighboring nodes of an element in a specified order (e.g. counter-clockwise). Based on the topology patterns reflected in a mesh, meshes can be classified into two main classes: structured and unstructured meshes.

Structured Mesh (Grid)

A structured mesh is one where "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" [4]. Simply put, in structured meshes, the interior vertices and elements are topologically alike and the neighbor connectivity can be implicitly induced. Figure 1.2 is an example of a simple structured mesh. A structured mesh is often called a grid. Structured meshes are commonly used because of their simplicity. Since the connectivity of structured meshes is implicitly assumed from the element shape, pattern, etc. the data structure used to store them is easy to access and and can save on storage requirements. However, it is difficult to use structured meshes to fit complicated geometric domains, which motivates the use of unstructured meshes.

Unstructured Mesh

An unstructured mesh is one "whose element connectivity of the neighboring grid vertices varies from point to point" [4]. Any mesh that is not a structured mesh is an unstructured mesh. An example of an unstructured mesh is shown in Figure 1.3. (The magnetron mesh in Figure 1.3 can be found at the following web-page: http://www.geuz.org/gmsh/gallery/magnetron1.gif)

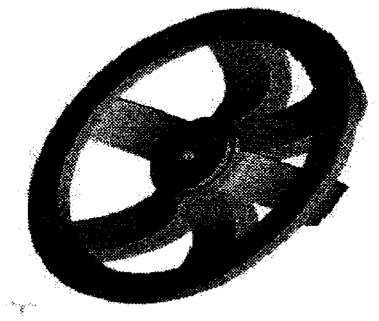


Figure 1.3: Unstructured Finite Element Mesh by P. Lafvre

Compared to structured meshes, unstructured meshes are more difficult to produce since the internal data structure is more complex. They also require more storage space since the connectivity information must be explicitly stored. However, unstructured meshes are often required by practical problems because they are capable of fitting complicated domains, and the quality of unstructured meshes can be improved by rapid cell grading and refinement and de-refinement [4].

1.1.2 Physical Mesh Information Used by Finite Element Analysis Programs

When a mesh generator is used as a pre-processor for a finite element analysis, the finite element program requires information related to the properties of the physical problem. This type of information is called the physical attributes. Three common types of physical attributes are boundary conditions, location and number of degrees of freedom, and material properties.

Boundary Conditions

Boundary conditions are physical conditions applied on the domain boundaries. Figure 1.4 illustrates a quadrilateral mesh of a 2D rectangular domain(dimensions L by W), which could be used for analysis of a solid mechanics problem to solve for displacements in the x and y directions(u_i and v_i, respectively). The boundary conditions on the domain are described in terms of applied tractions (*e.g.* τ_x), prescribed displacements (*e.g.* Δ_y), and fixity (*e.g.* roller versus pinned versus free). Two common types of boundary conditions are Neumann and Dirichlet. A Neumann boundary condition specifies the gradient of the dependent variable in a PDE, whereas a Dirichlet boundary condition specifies the prescribed value of the dependent variable.

Location and Number of Degrees of Freedom

In the finite element method the dependent variables of the PDE become degrees of freedom that are solved for at the nodes. Nodes may be located at the vertices in a mesh, but they can also be located at midpoints of an edge, the centroid of an element or elsewhere within the element. In Figure 1.4, the nodes are located at the vertices and each node has two degrees of freedom, one for displacement in each of the x and y directions. If there are more nodes than vertices in an element and the nodes are used to interpolate the geometry, the element can have curved edges.

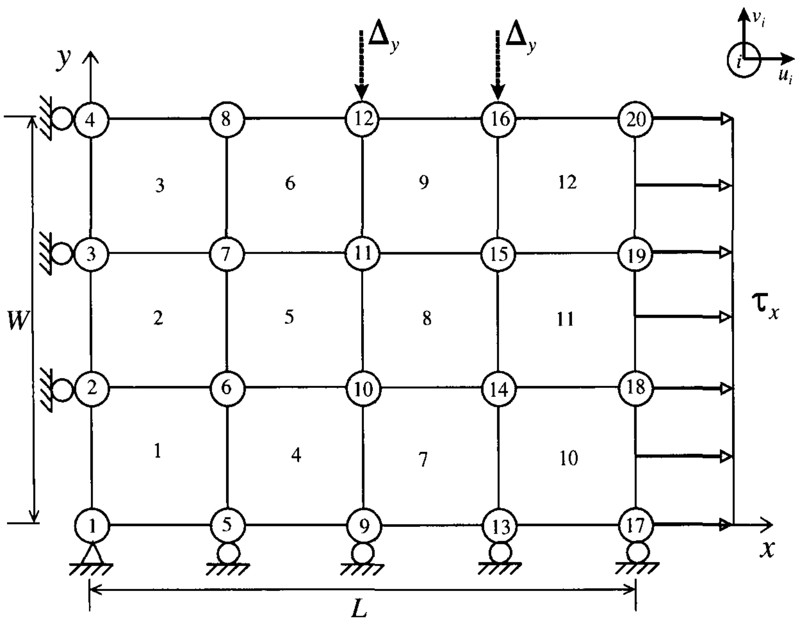


Figure 1.4: Example of a 2D Quadrilateral Mesh with Boundary Conditions

Material Properties

Material properties are another common physical attribute; they describe the material information for a domain. The entire domain may be of the same material, but it may also be divided into a number of zones, with each zone represented by a different material. In finite element analysis used for mechanical problems, common material properties include Poisson's ratio, Young's modulus, material density, etc.

1.2 Current Practices for Designing and Developing Mesh Generators

To evaluate mesh generator software, we take the viewpoint of a software engineer and thus the first question we ask is: How are today's mesh generators developed? We want to know whether there is an example of a mesh generator developed as a "software jewel" [42]? The term "software jewel" refers to "a well structured program written in a consistent style, free of kludges, developed so that each component is simple and organized, and designed so that the product is easy to change" [42]. We also want to know whether existing mesh generators have sufficient documentation to ease future extensions and modifications? This section is intended to answer the above questions by investigating current mesh generators. In Section 1.2.1, we evaluate a few representative example programs in detail to see how they have applied software engineering principles in developing their software. Section 1.2.2 summarizes the problems we observed in the existing designs.

1.2.1 A Look at the Design Strategies Used for Existing Mesh Generators

Based on the range of applicability, a mesh generator can be classified as either general-purpose or special-purpose. A general-purpose (GP) mesh generator is developed to solve a wide range of problems in many application domains. Most commercial mesh generator software is designed to be general-purpose; many software packages are capable of generating meshes on almost arbitrary closed shapes in 2D and 3D. Most GP mesh generators support advanced optimization features such as mesh refinement and grading. Special-purpose (SP) mesh generators, on the other hand, are those that are tailored to specific problems. For instance, a special-purpose mesh generator may target a certain problem domain, or it may be restricted to generate meshes on a simply shaped domain. Although general-purpose mesh generators provide considerable functionality, research scientists and engineers still write special-purpose mesh generators to solve their specialized problems. In such cases, limited options of a special-purpose mesh generator provide more convenience for solving a specific problem. For example, a special-purpose mesh generator for thermodynamic problems can fix the common physical attributes that are exclusively related to the intended problems, thus reducing unnecessary effort and reducing the complexity of the interface, which improves the quality of usability. In the same situation, a general-purpose mesh generator would require the same physical attributes be entered each time before generating the mesh.

There are at least hundreds, if not thousands, of mesh generators being developed and used today. A good source to acquire a list of available mesh generators can be found at [39, 45]. To help our investigation, we need mesh generators that are available in the public domain and provides sufficient design documentation so we can study them in detail. Based on these criteria, we have selected Qmesh Mesh Generation Package [29], Gmsh mesh generation software [20], and the AOMD library [24].

Qmesh Mesh Generation Package

QMesh [29] is a two-dimensional mesh generator designed for use with two-dimensional finite element analysis. Qmesh provides five programs, QMESH, RENUM, RENUM8, QPLOT, and QPLOTS. QMesh provides for a modular, flexible input method, whereby a body is described as a collection of regions. QMESH is the central program of the package, which reads the information about the input body and develop a mesh for each region independently and writes the mesh description to a file. The bandwidth minimization is performed by RENUM for four-node elements or RENUM8 for eight-node elements. QPLOT and QPLOT8 are plot programs to display four-node element meshes and eight-node meshes, respectively.

The organization of the package is based on the task to be performed (*i.e.* mesh generation, bandwidth minimization, mesh display). This is a poor decomposition approach because a design decision (*e.g.* file format) is embedded in many modules and a change in one module requires changes to other modules as well. As a result, the system is less changeable and reusable. Another undesired design is the existence of the similar programs such as QPLOT and QPLOT8. The two programs are basically performing the same task; therefore, they could be developed together by sharing common modules. However, the system is not designed in a way that a module can be reused for a subset of service offered by it. Instead, a separate program had to be written to provide this service.

Gmsh Mesh Generation Software

Gmsh [20] is an automatic 3D finite element mesh generator with a build-in CAD engine and post-processor. It provides a good meshing tool and advanced visualization capabilities to solve academic problems with parametric inputs.

According to the reference manual [20], the software consists of four modules: geometry, mesh, solver, and post-processing. Gmsh provides two ways to specify a boundary domain: the input values can be entered either via the graphical interface or via the scripting language. The boundary is specified in a bottom-up approach; primitive entities such as points and lines are specified first, followed by two-dimensional shapes (e.g. triangles, quadrilaterals) and three-dimensional objects (e.g. tetrahedral, hexahedra). The scripting language in Gmsh allows parametrization of the geometry so that a complex spatial domain can be constructed from basic geometry entities with relative ease. Unfortunately, Gmsh provides limited options for file format customization by only allowing the user to choose from a list of available file format extensions. The solver and post-processing modules in Gmsh integrate into a finite element solver, which works with the file format defined by Gmsh; therefore, the problem with customizing the file format is solved, but only in the case where the user decides to use the Gmsh solver.

When investigating the documentation in further detail, we found that the term "module" is used to group a set of data structures and functions. For example, the geometry module is a container of the geometry entities used by Gmsh (i.e. points, lines, circles, etc). However, the modules used in Gmsh contains too much information and the internal structure and module secrets are not explicitly explained in the documentation. Also, the documentation is written based on the flow of execution instead of modular structure; as a result, the relationship between components is unclear and we cannot decide whether reuse is possible during the system design.

AOMD Library

AOMD [24] (Algorithm Oriented Mesh Database) is a mesh management database aimed at effectively maintaining general mesh representations. The implementation language of

AOMD is C++ and the standard template library (STL) is extensively used to consistently store different types of mesh entities through generic data structures such as containers, iterators, etc. AOMD uses object-oriented programming for building a hierarchy of classes and uses the generic paradigm to build STL-like algorithms.

The structure of AOMD is well documented in its reference manual [46]. All the data structures in AOMD are defined as a base class using class templates. When a mesh is created, its entities are built through inheritance from the base class. If mesh entities are not static (e.g. nodes and elements that are added or removed), a hash table is used to store them so that the average time it takes to add and remove entities in a mesh is constant. Functions to implement and change a mesh are developed in the same way. By using generic concepts, the implementation of AOMD is "light and efficient in terms of compilation and memory use" [24].

Reuse is clearly emphasized in the implementation of AOMD through class inheritance and template initialization. One of the design goals of AOMD is to achieve algorithmic generality such that the algorithm can be used regardless of the concrete mesh representation. This is very close to our research objective. Therefore we can use its design for reference in the future. However, mesh generation with AOMD could be improved with respect to software usability. For instance, the use of a graphical user interface would be more user-friendly than the text-based interface provided by AOMD. In addition, AOMD does not provide an automated approach to developing different members of a program family, and the programming task to generate a mesh is often more complicated than necessary for simple problems. In this case, an automated approach to generate mesh generators is desired and when usability is a requirement, a graphical user interface (GUI) with menu-based instructions is a much better solution than a text-based interface.

1.2.2 Problems with the Current Approach

After reviewing today's literature on mesh generation, we had the following general observations. First, most papers related to mesh generation focus on the algorithms or methods for the mesh generation technique, and there are only a few documents on the software design approaches that have been taken, such as [5, 17]. The lack of publications on mesh software design is illustrated by the fact that among the approximately 120 papers available on Meshing Research Corner [39] from 2002 to 2004, there are only three papers [7, 19, 21] that talk about the software design of mesh generators. Secondly, although many special-purpose mesh generators are used today, a research effort on their systematic development is still lacking. A problem with developing special-purpose mesh generators is that most of them are only developed when they are needed to solve specific problems, thus ignoring the possible connection between programs. Our observation concludes that the software engineering principles are rarely considered and followed in the field of mesh generation, and the lack of such practice has caused the following problems:

[1] Many mesh generators do not encourage program reuse. For example, in Qmesh, the existence of two mesh plot programs is unnecessary. We can reuse QPLOT8 for the four-node element mesh as well. This problem indicates there is a lack of software engineering practice in the field of mesh generation and scientific computing. The involvement of software engineers in these fields would greatly benefit the development of mesh generators and other types of scientific software.

[2] The documentations of many mesh generators are often incomplete, ambiguous, or even non-existent. For instance, the documentation for GRUMMP [38] consists only of a user manual and the source code, even though GRUMMP has a complex object-oriented design. Without clear documentation, it is difficult to understand the software structure. A lack of documentation also matters when it comes to future maintenance.

[3] The development of similar mesh generators lacks a systematic approach. The fact that many existing mesh generators are similar in terms of mesh types, interface requirements, etc. has led us to believe that it is advantageous to develop them as a family. These mesh generators can be developed by reusing the common aspects

developed in advance, and considering their differences systematically.

[4] The mesh generation for specialized problems is not addressed. For example, a thermodynamics parametric study may require repetitive meshing of a simple rectangular domain of a heat plate. Although specialized problems can be solved with general-purpose mesh generators, we argue that this is not the most efficient solution. First of all, general-purpose mesh generator provides too many details that may distract the engineer from their relatively simple problem. For instance, if the problem always needs to discretize a rectangular domain, then a package that support mesh generation on arbitrary closed shapes is more complicated than required. Secondly, the interface of general-purpose mesh generators is too complicated for simple meshing problems. For instance, Gmsh requires 15 steps to specify a rectangular domain and generate the mesh. If the same problem arises repetitively, the use of Gmsh is inefficient because the user has to repeat the same multiple sequence many times.

[5] Most general-purpose mesh generators that we studied offer little or no customizing capability for output file(s) at run-time. Although software such as GRUMMP [38] allows user-defined formats to customize the output file(s), the same degree of customizing capability is rarely implemented in other mesh generators. Our literature research reveals that most GP mesh generators take a common approach by providing the user with a set of pre-defined formats in which the output file(s) can be stored. For example, Gmsh defines only one native file format with extension *.msh*, along with providing several other common export formats. The previous approach allows the user to customize the output file(s) by choosing the format he/she requires. However, this type of customization control offers very limited flexibility. For example, if the user wants an output format which is not provided by the mesh generator, he/she must write a program to transform the output file to the format they require. A better solution to this problem is that if the user knows the input format for the finite element program for which the mesh is used, then he/she should be to able to cus-

tomize the format of the output file(s) through the mesh generator interface before generating the mesh.

Our research goal is to come up with a systematic approach to develop mesh generators for special purpose problems. It is a fact that these special-purpose mesh generators are similar in many ways. For example, they deal with the computational domain bounded by a relatively simple shape, and generate two-dimensional or three-dimensional elements, etc. Software engineering experience tells us that if we are developing a set of related programs, it is advantageous to consider them as a family where the common properties (*e.g.* graphical interface, mesh type) can be shared by all family members and the distinctions among them (*e.g.* file format) can be explicitly identified.

1.3 A Review of Program Family Methodology and Family-Oriented Software Production

A continuing dilemma that today's software engineers face is the objective of producing functional programs that are designed for flexibility, while under time constraints for doing so. A solution to this problem is the concept of program families [40, 41]. This section is aimed at providing fundamental knowledge about program family design. Section 1.3.1 explains why mesh generators fit in the context of program families. Section 1.3.2 introduces several design techniques that can be used to achieve the goals of producing program families and generating reusable components.

1.3.1 The Appropriateness of a Mesh Generator Family

Parnas said, "When we were first taught how to program, we were given a specific problem and told to write one program to do that job. Later we compared our program to others, but still assuming that we were producing a single product" [41]. However, when it comes to today's software development, one software designer must be aware that he is often not designing a single program but rather a family of programs. Today's software products must undergo frequent changes to meet new requirements and to improve quality. With limited time, it is almost impossible to go through redesign, recoding, and retesting to create a new individual system to accommodate changes. In this case, creating program families is a much better solution because this approach support rapidly producing new family members with little or no redesign, recoding, and reduced retesting. Parnas defines a program family as "a set of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members" [40]. The concept was first proposed by Dijkstra [15], and later investigated by Parnas [40, 41]. It has also been applied by Weiss [2, 53, 54] in the context of what he calls Family oriented Abstraction, Specification and Translation (FAST) [13, 52].

The suitability of mesh generators as a program family has been argued by Chen [8] and Smith and Chen [47], because the development of mesh generators satisfies the three hypothesis for the production strategies of a program family as introduced by Weiss [53]. First, the redevelopment hypothesis states that most software development involved in producing the family should be redevelopment work. In the case of mesh generators, redevelopment is common such that most small code are based on modification of existing code. Second, the oracle hypothesis requires that the likely changes in the future of the system's lifetime be predictable. This is certainly the case for a mesh generators where one can determine likely changes by consulting the large body of literature and mathematical theories on the topic. Thirdly, the organizational hypothesis says that a program family should allow the developers to organize the software in the way that changes can be made independently. In mesh generators, the changes in such areas as the user interface, visualization, and output format can be dealt with in an elegant manner. Even changes in the mesh algorithm, which are complicated by the coupling between the data structures and the algorithms, can potentially be made independently of other modules in the program [17, 24].

Mesh generators fit Parnas's definition of a program family because they possess a set of common features. For example, they are similar in that all mesh generators require a computational domain and then discretize it, and they all produce output file(s). On the other hand, the distinctions between mesh generators can be identified as well. For instance, different meshing programs use different mesh generation algorithms, and produce output files in different formats. By identifying the commonalities and variabilities of mesh generators, we can rapidly produce programs by reusing the code common to all family members, and then adding the customizing capability as the variabilities.

The advantages of developing mesh generators as a program family is obvious: If the common features of mesh generators can be identified and the variabilities predicted, the mesh generator production can be systematic. As a consequence of systematic production, the cost of program development and maintenance can be reduced. The conflicting goals of rapid production and careful engineering can be satisfied concurrently [2]. However, this practice requires significant investment and effort in identifying the commonalities and variabilities among family members. As we will point out in the next chapter, a commonality analysis document provide a systematic methodology for the identification process.

1.3.2 Current Techniques of Family-Oriented Software Development

The key in family-oriented software development is the prediction of future changes. A well-structured design, that is, a design that can be easily understood and, most importantly, easily modified can become extremely useful when change needs to be accommodated. The program family methodology facilitates a well-structured design since it satisfies the organizational hypothesis; that is, changes among family members can be predicted in advance and made in only a few modules. The practice of family design has been implemented for FAST [52], and general implementation strategies have been formulated [16]. This section introduces specific techniques for generation of family members or reusable components. The use of configuration constants, conditional compilation, generic programming, and domain-specific languages will be discussed, respectively.

Configuration Constants

Configuration constants are symbolic constants provided in programming languages to make programs easily adaptable to modifications and generate new program family members [16]. One problem with software modifications is that if specific information that may change is spread throughout the program, then changes need to be made to each reference to that information in the program. For example, if an array is declared to be of size 10 and later compared with an integer variable for boundary checking, then any change to the size of the array requires changing both declaration and boundary checking statements. In this case, if the required changes are a set of configuration constants, the problem may be solved by changing the value of such constants and then recompiling the program. The use of configuration constants is a simple solution to implement a program family. The configuration constants also have the advantage that they can be given names that suggest their meanings, which improves readability and modifiability of the program. However, configuration constants only works when the distinction between family members can be represented by constants. When the difference between family members becomes more complex, a more general approach such as conditional compilation is needed.

Conditional Compilation

Conditional compilation provides a more flexible and general scheme to implement a program family and generate new family members [16]. The essence of conditional compilation is that all family members are produced from one copy of source code, and the difference between family members is identified by a precondition in the code. When the program is compiled, the source fragment common to all family members will be included, along with a variable fragment if the corresponding precondition is satisfied. Each compilation produces only one program because the code that is not part of the specified fragment will be ignored by the compiler. Conditional compilation includes configuration constants, as they are used in the precondition during the execution of the conditional compilation.

Generic Programming with Templates

Generic programming is a programming paradigm aiming at code reuse and abstracting general concepts. The term *generic* means to create code that is type-independent. Generic programming extends the language so that one can write a function for a generic type once, and use it for a variety of actual types. In a program family produced with generic programming, the algorithms are considered as the commonality of all members of the family, and the differences between them are the concrete data types that are instantiated. For example, a sorting algorithm can be written to perform a common sorting procedure for all family members, each of which will provide a different data type (*e.g.* integer, floating point, etc) for its own implementation purpose. A programming language that support general-programming well is C++. The C++ templates and the Standard Template Library (STL) provide a good mechanism to write type-independent code for functions, pointers, and classes.

Software Generation through Domain Specific Languages

A domain specific language (DSL) is a small, usually declarative language, that offers expressive power focusing on a particular problem domain. A DSL can be viewed from two perspectives: a programming language or a software architecture [10]. A DSL is a programming language, or an executable specification language, that offers convenience and flexibilities in software design and implementation. This convenience makes a DSL more attractive compared to general-purpose languages (GPL) such as C and C++, when programming for a specific domain. For example, DSLs provide easier programming environments than GPLs. A program written in DSL is more concise and readable than its GPL counterpart because of appropriate abstractions, notations and declarative formulations. Another favorable property of DSLs is that they promote systematic reuse by integrating common operations into libraries. DSLs also offer built-in functionalities to encourage re-use.

From the software architecture perspective, a DSL addresses the important issue of a program family. As Consel [10] says, "a DSL program designates a member of a program family." If it is foreseen that the program to be developed addresses a set of related problems, it is worthwhile to adopt the program family methodology and design a DSL from which family members are developed.

There are different ways of using DSLs in a family-oriented design. First of all, a DSL can be used to generate executable code to produce family members. In this approach, the DSL can be termed as an application modeling language (AML) [41] that provides a specification of each family member from which the deliverable code for the family member may be generated. The FAST process proposed by Weiss [13, 52] provides facilities that typically consist of a language for specifying family members, a translator for generating a family member from a specification in the language, and tools for analyzing such a specification. An application of the FAST process is a floating weather station family that uses a DSL to configure buoys [52]. Secondly, a DSL can be used as a specification for run-time customization. One example is XML (Extensible Markup Language) data binding where the DSL is developed as a XML document and the binding of the document is to customize an Java object. XML is a structured language widely used for knowledge modeling in many scientific fields, such as mathematics [11], chemistry [37],and material study [22]. XML data binding refers to the automatic generation of computer language source code corresponding to an XML DTD or schema. Recent research has developed tools, such as JAXB (Java Architecture for XML Binding) [35], to provide a convenient way to bind XML data to Java code and objects [34]. Thirdly, a DSL program can be developed as input to a custom compiler developed by tools such as Lex and Yacc [26, 33]. Lex [33] is a lexical analyzer generator designed for lexical processing of character input streams. It accepts a high-level specification written in a DSL language for string matching and processing. The core of Lex is a table of regular expressions and corresponding program fragments, which are used to translate the input streams and partition them into strings that match the regular expressions. If a string is recognized in the DSL source program, the corresponding code fragment will be executed by Lex. A Lex program can be written in different languages, such as C, to generate the output code to process the DSL program. Yacc [26] stands for "yet another compiler compiler" and provides a tool for imposing structure on the input to a computer program. To use Yacc, a user prepares a specification which includes a collection of rules to describe the input structure, code to be invoked when the rule is recognized, and a low-level routine provided by Lex to process the basic input. Yacc then generates a function called parser to control the input process. The generated parser is responsible to read the source program, which is the output produced by Lex, and perform the actions as specified in the specification. Lex and Yacc combined offer a programming environment for developing customized compilers to execute a DSL program developed to meet the users' demand. They have been extensively used in numerous practical applications, such as lint [27], the portable C compiler [28], and a system for typesetting mathematics [31].

1.4 Purpose and Scope

As mentioned previously, our research interest is in the type of mesh generators used as a pre-processor to a finite element program. Having identified the problems that we observed in the current development of mesh generators, it is clear that there is a lack of a software solution for rapidly developing special-purpose mesh generators. In our thesis, we fill this need by proposing a technique for automated generation of a program family of special purpose mesh generators. Our goal in this thesis is to create a type of software that not only overcomes the difficulties we have discovered but is also developed with a systematic approach to achieve important software qualities, such as correctness, user friendliness, reusability, maintainability, and portability. It is also important that the software should be well-documented and any future changes to the software should be consistent with its documentation.

As mentioned previously, the mesh generators of interest in this thesis are those used as pre-processors for finite element programs. From the mesh generation perspective, we only focus on the mesh generators that generate 2D structured planar meshes with triangular or quadrilateral elements. The coordinate system of the mesh will be Cartesian. From the standpoint of software engineering and automated program generation, our scope includes a design of a DSL in XML format, XML run-time binding for Java software generation, a graphical user interface for all family members, and the use of XSL stylesheets to produce customized output file(s).

1.5 Overview of the Thesis

In Chapter 2, we will present our methodology for developing a program family generator. This chapter covers the phases of commonality analysis, gathering and analyzing software requirements, and system architecture. The documentation, including commonality analysis (CA), software requirement specification (SRS), and module guide (MG) will be discussed in the relevant appendix sections.

After completing the software design, we move on to Chapter 3 to explain how our system was implemented. This chapters starts with design decisions related to our implementation such as the DSL design and the file format transformation. The rest of the chapter follows the order in which the implementation proceeds. Starting from the design of our DSL, we will explain the mesh seed information specification, mesh generation and its algorithms, and the use of XSL stylesheet for output file customization.

To convince the reader that our system is practical, we will use Chapter 4 to provide case studies to solve real problems. This chapter serves as a user guide to give an overview of how to use our system. Starting from system initialization, we will illustrate each step in how to generate a mesh generator with our product. In Chapter 5 we will conclude our thesis by first evaluating our approach, followed by a summary of the contributions of our work. Finally, we end the thesis with a list of possible future research directions. The appendices are documents that we have produced during our design, including a CA, an SRS and an MG. A test plan is also included in the appendix.

Chapter 2

Software Requirements and Design

After proposing our research goal, which is to support rapid development of a program family of special-purpose mesh generators, our attention now turns to accomplishing this goal. The traditional waterfall model [16] suggests that the lifecycle of a software system consists of five phases: requirements analysis and specification, design and specification, coding and module testing, integration and system testing, delivery and maintenance. With the exception of delivery and maintenance, which is outside the scope of this thesis, each phase will be discussed in this chapter. In family-oriented software development, we must also identify the family before performing requirement analysis. This is done by conducting a commonality analysis, which systematically identifies the similarities and differences among family members.

We demonstrate an engineering approach to designing a program family of special-purpose mesh generators. Section 2.1 introduces the notion of commonality analysis and discusses how a suitable program family of mesh generators can be identified. Section 2.2 identifies the requirements for the mesh generator family we intended to build, and briefly presents our requirement specification. The module guide will be provided in the Appendix C.

2.1 Commonality Analysis

In Chapter 1, we discussed the reasons why it is appropriate and possible to develop a program family of mesh generators. The success of developing a program family depends on how well the software engineers can predict what family members will be needed [40]. In family-oriented software development, a systematic methodology to identify a program family is a commonality analysis.

A commonality analysis is an analytical approach that consists of systematically identifying and documenting the commonalities that all program family members share, the variabilities between family members and the terminology used in describing the family. It is initially proposed by Weiss [40] and has been demonstrated in FAST [13, 52]. A commonality analysis provides a systematic way of gaining confidence that a family is worth building and of identifying what the family members will be [48]. According to Weiss [40, 41, 48], a commonality analysis can benefit the system design in the following ways: a starting point for the design of DSLs, a basis for a common design for all family members, a historic reference for future improvement, a basis for reengineering a domain, and a basic training reference for future software developers. The commonality analysis can be used as the starting point for writing a requirement specification.

In Weiss's proposal [40, 41], a commonality analysis document consists of five sections. They are listed in the following order:

[1] A list of terminologies and definitions used in the document.

[2] A set of commonalities that represent the common features that all family members share.

[3] A set of variabilities that distinguish family members from one another.

[4] A set of parameters of variation, where a parameter of variation is the range of values that can be assigned to the corresponding variability.

[5] A list of issues that arose during the commonality analysis, but that do not fit in the other sections.

In the field of mesh generation, a commonality analysis for 2D mesh generators was documented by Chen [8], and later revised by Smith and Chen [47] to cover all mesh generators that are used as finite element pre-processors. Since we are interested in finite element pre-processors, we will use the templates and examples from Smith and Chen [47] as the basis of our commonality analysis. For convenience of reference, we will use S&C as an abbreviation for Smith and Chen [47]. The commonality analysis described in this chapter is documented in Appendix A.

The remainder of this section discusses a commonality analysis for the mesh generators used as pre-processors for finite element programs. Section 2.1.1 introduces the concept of commonalities and explains some example commonalities for mesh generators. Section 2.1.2 presents the variabilities of mesh generators, and further elaborates specific examples. Section 2.1.3 discusses the parameters of variation, and demonstrate how the variabilities in mesh generators can be quantified by assigning different values to the corresponding parameters of variation.

2.1.1 Commonality

A commonality is "a requirement or goal common to all family members" [47]. The concept of commonality captures the common features that all members of a family share. In the domain of mesh generators used as pre-processors for finite element programs, some common features that are shared by such mesh generators are easy to find. For instance, some of common features listed in Appendix A are as follows:

[1] A mesh generator discretizes a given computational domain into a covering up of a finite number of simple shapes.

[2] Each vertex has a unique identifier.

[3] Each element has a unique identifier.

Item Number	C1
Description	A mesh generator discretizes a given computational domain (closed boundary Ω) into a covering up of a finite number of simple shapes.
Related Variability	V4, V7, V8, V10
History	Borrowed from S&C on May 9, 2005

Table 2.1: An Example Commonality

[4] Information on the created meshes includes boundary conditions.

[5] A mesh generator requires that information be input by the user to define his/her meshing problem.

[6] The user needs to specify the physical attributes, such as the material properties, the boundary conditions, etc.

[7] Mesh generators write mesh information to output file(s).

In a commonality analysis document, the commonalities should be documented in a uniform structure. In S&C and Appendix A, each commonality is described in a table with four fields. The structure of the table is as follows: First, each commonality is given an identifier prefixed by a capital "C" followed by a unique natural number. Second, the description of the commonality is provided, followed by a set of related variabilities. Finally, each commonality ends with a summary of its history, including the date of creation and any dates of modification, along with a brief description of the modification. An example commonality is provided in Table 2.1.

As discussed in Section 1.4, our research scope is restricted to structured mesh generators generating 2D planar meshes, etc. Therefore, our commonality analysis covers a narrower range of mesh generators than those covered in S&C. As a result, new commonalities are added by fixing the parameter of variation for some of the variabilities in S&C.

For example, if we only consider structured meshes, then the variability that allows unstructured meshes in S&C should be changed to a commonality that states all program family members generate structured meshes. Similar changes also need to be made to several other variabilities, as shown in Table 2.2. Each item in Table 2.2 consists of new commonalities added in our commonality analysis document. In Table 2.2, all references to "V?", where "?" is a natural number, are referring to variabilities documented in S&C.

Identifier	**Commonality Description**	**Old Variability**
C8	The mesh generators will not provide optimization features such as smoothening, and refinement/coarsening.	V3
C11	From topological perspective, all mesh generators in our scope produce only structured meshes.	V6
C12	The local numbering of vertices and nodes in the mesh is always counter-clockwise.	V8
C13	The mesh generators should not accommodate a mixed mesh, a hybrid mesh, or a nonconformal mesh.	V16–V18
C14	The mesh generators only use Cartesian coordinates to describe the geometry.	V20
C16	The mesh generators provide graphical user interface.	V21
C17	The interface for specifying the closed boundary of the domain is provided as the number of subdivisions along each direction.	V22
C18	The mesh generators will not allow the specification of two degrees of freedoms to have the same value.	V28
C19	The mesh generators will not allow the specification of internal boundaries.	V29
C20	The user defines the geometric domain of the problem by parametric representations.	V23
C23	The mesh generators display the resulting mesh on the screen.	V30

Table 2.2: Modifications to Commonalities

2.1.2 Variabilities

A variability is “a requirement or goal that varies between family members” [47]. The set of variabilities shows how family members can be distinguished from one another. In the case of our intended mesh generators, the variabilities can be seen from the different types of meshes that can be produced. For example, mesh generators for 2D domains differ in the type of element shapes they can produce. Another example of variability is the format of the output files produced by the mesh generators. A sample list of variabilities between mesh generators as documented in S&C. These variabilities are also variabilities for the current project, are as follows:

[1] Different mesh generators will be able to accommodate the creation of meshes for different problem domains (e.g. solid mechanics, thermodynamics).

[2] For structured meshes different templates for the local patterns in the element topology are possible.

[3] The number of different materials allowed in the specification of the physical problem.

[4] The format of the information in the file(s) output by the mesh generator.

[5] The degree to which the user can customize the output file formats.

In S&C, the template used for documenting variabilities is similar as that used for commonalities. Each variability is described in a table with five fields. The structure of the table is as follows: in the first row, each variability is given a unique identifier prefixed by a capital “V” followed by a natural number. Second, the description of the variability is provided, followed by a set of related commonalities, which can be used to refer back to the previous section of commonalities. Third, the parameter of variation is provided to

Item Number	V7
Description	For structured meshes different templates for the local patterns in the element topology are possible.
Related Commonality	C1
Related Parameter	P7
History	Created - May 9, 2005

Table 2.3: An Example Variability

show how the variability is quantified for all family members. Finally, each variability ends with a summary of the history, including the date of creation and any dates of modification, along with a brief description of the modification. To illustrate a variability in further detail, we pick an example of a variability documented in our commonality analysis as shown in Table 2.3.

Since we restrict the scope of research by only considering the type of mesh generators presented in Section 1.4, there will be a reduction in the number of variabilities, which are now listed as commonalities in our commonality analysis. The variabilities listed in S&C that were removed are shown in Table 2.1.

2.1.3 Parameter of Variation

A parameter of variation is a way of quantifying variabilities by assigning the possible values to the family members that reflects their differences. In commonality analysis, each variability is identified by the corresponding parameter of variation. For example, the possible types of element shapes in 2D domains are quadrilaterals and triangles. A sample list of parameters of variation between mesh generators as documented in S&C, and which are also parameters of variation for the current project, are as follows:

[1] Mesh generators can be built either to be general-purpose for an arbitrary range of

problem domains, or to be special-purpose for a specialized problem domain, such as solid mechanics, structured fluid dynamics, heat transfer, *etc.*

[2] The possible element topology patterns are listed in Appendix A.6 in the referenced document.

[3] The entire domain can consist of one material or there may be any finite number of different materials.

[4] Different mesh generators organize mesh information into file(s) in different orders. The data structure that is output can change between mesh generators.

[5] Some mesh generators will have a fixed file format, while others will allow the user to customize the output.

In S&C, each parameter of variation is documented in a table of four rows. The table is constructed with the template as follows: first, each parameter of variation is assigned a unique identifier of the form "P" followed by a natural number. Second, the related variability is indicated so that the parameter of variation can be traced back to the corresponding variability. Third, the range of parameter field enumerates the range of possible values for all family members. Finally, the binding time indicates the time that the variability is fixed. The value of a parameter of variation can be fixed during specification (specification time), or during building of the system (compile time), or during execution of the system (run time). It is possible to have a mixture of binding times. For instance, a parameter of variation could have a binding time of "specification or building" to represent that the parameter could be set at specification time, or it could be postponed to the time when the family member is built. The choice of postponing the decision until the build would be associated with the presence of a DSL that would allow postponing decisions on the values of the parameter of variation. If the value for a parameter of variation has been fixed, the

Item Number	P7
Corresponding Variability	V7
Range of Parameters	Nine (9) potential local topology templates are possible, as shown in Appendix A.
Binding Time	specification or build or run time

Table 2.4: An Example Parameter of Variation

different values of its binding time would still constitute different family members in terms of generality. For example, a mesh generator fixing the geometry of a domain at run time is more general-purpose than another mesh generator fixing the same value at specification time.

To illustrate a parameter of variation in further detail, we pick an example of parameter of variation documented in our commonality analysis. The structure of documenting such a parameter of variation is provided in Table 2.4.

This parameter of variation is given to show what patterns may be used for structured meshes in our research. In a mesh generator, the value of such a pattern may be fixed at specification time or build time. The use of a DSL in our implementation provide the convenience to parameterize the value of the interested pattern at specification time. Another option is to postpone the decision until run-time, where the mesh generator is already built and the pattern can be specified in the user interface before generating the mesh.

Besides removing parameters of variation from S&C for those variabilities that become commonalities (Table 2.1), another type of change is made to some parameters of variation, as shown in Table 2.5. These changes involve modifying the original parameters of variation to reflect our scope. Please note that all references to "P?" in Table 2.5 refers to parameters of variation in our commonality analysis as shown in Appendix A, not to S&C.

Identifier	Old Description	New Description
P7	Seven(7) potential local topology patterns are provided in Appendix A of S&C.	Nine(9) potential local topology patterns are provided in Appendix A of our commonality analysis document.
P9	In 1D there are line segments; in 2D there are triangles and quadrilaterals. In 3D, there are tetrahedras and hexahedras.	In 2D there are triangles and quadrilaterals.
P13	1D, 2D, or 3D	2D
P14	The computational domain in 1D can be either a straight line or a curve. For 2D and 3D the domain can consist of simple shapes, such as triangles (tetrahedra), rectangles (boxes), parallelograms (parallelepipeds), etc.	In 2D domains, the possible shapes are those accepted by Zienkiewicz and Phillips.
P19	1D, 2D, or 3D	2D

Table 2.5: Changes Made to Parameter of Variation Between S&C and the current study

2.2 Requirements Specification

The likelihood of a good software design increases when a software designer has a specification of the system's required functionalities and qualities. To determine the software requirements, the designer works with and decides what the software should do to meet the users' demand. This is the process of gathering, analyzing and documenting software requirements. As indicated by the waterfall model of software development, a rational design process should start with establishing and documenting software requirements. A successful software design process must be accompanied by its software requirement specification (SRS), which contains everything the designers need to know to help write software that is acceptable by the user.

In this section, we look into the details of this crucial stage of software development. We will first introduce the concept of software requirements. Then, we will discuss the approaches we took in identifying the specific requirements of our system, and highlight the keys in our requirement gathering process. Finally, we will list specific requirement examples from the SRS provided in Appendix B.

2.2.1 What is a Software Requirement

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" [50].

An important quality for good requirements is that they be abstract, where abstract means not imposing design decision; that is, the requirements should be about "what" is required as opposed to "how" to do it. It is important to recognize the boundary between the statement of software requirements and how the requirements will be implemented. For instance, a word processing application may have a requirement to be able to open an existing file. However, it is a design issue whether to build a customized file selection tool or to use a platform-standard file selection tool to satisfy the requirement.

There are several types of software requirements. We categorize them into three types: functional requirements, non-functional requirements, and system constraints. Each type of requirements is listed in separate sections in the SRS.

Functional Requirements

A functional requirement describes the functionality that the system or a component of the system must possess. In other words, a functional requirement should answer the question "what is the system supposed to do?" The set of functional requirements capture the essential behaviors of the system and must be stated precisely to avoid any ambiguity. The previous example of a word processing application is an example of a functional requirement.

Non-functional Requirements

A non-functional requirement describes the property that the system as a whole must possess. Non-functional requirements deal with software quality issues such as performance (e.g. speed, availability), portability, maintainability, usabilility, security, etc. When describing a non-functional requirement, one should be specific and use quantitative measurements so that the requirement can be validated. For instance, to describe the performance of opening an existing file, one can say "opening a file should take less than 3 seconds for 90% of the files and less than 10 seconds for every file."

System Constraints

A constraint is a "factor that is imposed on a solution" [1]. A system constraint puts a restriction on external environments in which the system operates or the system may be interfacing with (e.g. hardware, software). For example, a system constraint for the word processing application could be: "The application must operate on both the Windows and Linux operating systems." A system constraint may also be related to the project, such as the availability of resources assigned for the project, and the deadlines imposed for the delivery of the system. Constraints should be clearly stated in the requirement documents, since they directly influence the decisions of how the system is implemented.

2.2.2 Identifying Our Requirements

For our research, an SRS for our system has been written and included in Appendix B. In the subsequent sections, we want to highlight the two key steps in our requirement gathering process. First, we must provide a clear understanding of the system. Second, we have mentioned earlier that the commonality analysis for mesh generators can be modified to express requirements. In this section we provide some examples to show how this modification process was done in our research.

2.2.2.1 Understanding the System

Our intended system automates the generation of special-purpose mesh generators. Therefore, we are not building mesh generators, but a program generator that builds a program family of mesh generators. The system is called a "Mesh Generator Generator"(MGG). The mesh generation is performed after the mesh generators has been built. If one wants to construct a specific mesh for a general class of related meshing problems, two step are necessary: 1) Generate a mesh generator M using MGG, which meets the specification of the class of related problems, and 2) Generate the specific mesh using M. We will explain these two phases in further detail in subsequent paragraphs. Please note that in the requirement phase, we will not discuss the design decisions related to future implementations. The discussions of design decisions will be postponed until Chapter 3.

Generating a Family of Mesh Generators

The program generator that generates a mesh generator contains two parts: 1) the DSL source language and 2) the translator that builds mesh generators using the DSL program. As mentioned in Chapter 1, a DSL is a language that captures the essential set of information related to a specific field. In the case of mesh generation, the DSL contains all necessary mesh-related inputs to produce a mesh. For our research, the source of the required inputs is the commonality analysis, where the input section of the document clearly defines the information needed by a mesh. For example, a generated mesh contains geometry information as well as physical attributes. Therefore, the DSL must be able to represent both types of information. Other information may include system parameters, output file formats, etc.

The primary language construct used by the DSL is a property list, where each property is a specific mesh-related input mentioned earlier, such as material property, boundary condition type, etc. A property is expressed as a property name and a value for the property. For a specific property, which is introduced as a variability in the commonality analysis, its value is chosen from the corresponding parameter of variation. The binding time of a property value is given either as specification time or run time. Binding at specification time means the value is specified in the DSL program and cannot be changed later. The user decides what property values are bound at specification time and adds them into the property list, which becomes the DSL program. The DSL program is read by the translator and a mesh generator member is generated. A family of mesh generators can be built by specifying different values and binding times for properties in the DSL language. The degree of generality for a special-purpose mesh generator depends on the number of property values assigned in the DSL program; a mesh generator is more special-purpose than another if it is generated from a DSL program with more properties bound at specification time. Therefore, the most general-purpose mesh generator in our program family is the one generated from an empty DSL program.

Mesh Generation Using the Mesh Generator

Once a mesh generator has been built, the values specified at specification time are built into the mesh generator. Before generating the mesh, the property values missing from the DSL program must be specified in the user interface. This is called run time binding. The produced mesh is visualized in the user interface and the output file(s) stored in the user-defined format can be used as input by a finite element program.

2.2.2.2 Modifying Commonality Analysis Document as System Requirements

In Section 2.1, we introduced the commonality analysis and explained the importance of practicing commonality analysis in family-oriented software development. It has been clearly explained that commonality analysis can be modified to software requirements because commonalities and variabilities are in fact describing the desired behaviors between individual members of a program family. Therefore, they are requirements. More specifically, the commonalities are the requirements for all members of a family, whereas the variabilities are the requirements for different family members with the value of the difference quantified as the corresponding parameter of variation. However, since the commonality analysis document in Appendix A are only aimed at mesh generation, it can only be used for the requirements of mesh generators after they have been built. For a complete requirement document for our intended system, the requirements for the program generator must be considered as well. Table 2.6 provides the outline of the SRS documented in Appendix B.

1 Introduction
1.1 Purpose of Document
1.2 Terminology and Definitions

1.3 References
1.4 Overview
2 System Description
2.1 System Purpose
2.2 System Scope
2.3 System Context
2.4 Major System Capabilities
2.5 User Characteristics
2.6 Assumptions and Dependencies
3 Specific System Requirements
3.1 Functional Requirements
3.1.1 MGG features
3.1.2 Input
3.2 Nonfunctional Requirements
3.3 System Constraints
4 Other System Issues

Table 2.6: Outline of SRS in Appendix B

The connection between commonality analysis and the requirement specification can be found in the section of input requirements. Each input requirement is either associated with a commonality or a variability previously described in the commonality analysis. To explicitly indicate how the input requirements are modified from the commonality analysis document, we provide for each input requirement the item identifier from the commonality analysis in Appendix A. In particular, the corresponding parameter of variation and binding time are also given for the input requirements related to variabilities in Appendix A. We also labeled some commonalities as scope-time decisions because these commonalities were modified from variabilities in S&C to reflect the change of the scope.

The requirements of the program generator are beyond the scope of the commonality analysis, thus, they are established separately. A few detailed examples are given below. Table 2.7 and 2.8 are two example requirements for the MGG. Table 2.9 and 2.10 are example of a nonfunctional requirement and a system constraint, respectively.

Requirement ID	R1
Description	After the program generator reads the DSL input specification, it should either produce a mesh generator if there is no error in the DSL specification, or halt.
Commonality reference	–
Parameter of variation	–
Binding time	–

Table 2.7: Example of MGG Requirement

Requirement ID	R3
Description	The system MGG should provide the functionality to incorporate a graphical user interface for all family members.
Commonality reference	V21
Parameter of variation	All mesh generators use a graphical user interface.

Binding time	scope time

Table 2.8: Another Example of MGG Requirement

Requirement ID	R28
Description	The MGG should perform its essential functions in reasonable time, such as instantaneous for entering user input, seconds for waiting graphics display, or minutes for mesh generation and output file creation.
Commonality reference	C15, V39
Parameter of variation	–
Binding time	scope time

Table 2.9: Example of a Nonfunctional Requirement

Requirement ID	R36
Description	The first version of MGG is designed to run under Windows and MacOS because of the time constraint. However, we intend to develop the future versions to run under multi-platforms such as Linux.
Commonality reference	C19, V37, P37
Parameter of variation	–

Binding time	scope time

Table 2.10: Example of a System Constraint

Chapter 3

System Implementation

As we mentioned in Chapter 2, we intend to build a system that facilitates automatic generation of a program family of mesh generators. In the FAST design process [13, 52], Weiss proposes an application development environment, which provides facilities that typically consist of a language for specifying family members, a translator for generating a family member from a specification in the language, and tools for analyzing such a specification. Since our research goal is also an application of producing program family members, the implementation of the system follows a similar approach as suggested in FAST. This chapter introduces the design decisions that were made prior to coding, and it provides details about the implementation stage.

The chapter is organized following the order of each system module is implemented. Section 3.1 discusses the adoption of XML to implement the DSL to model the requirements of mesh inputs. Section 3.2 focuses on the meshing algorithm used by all mesh generator family members. Finally, Section 3.3 discusses the use of XSL to customize the format of output file(s).

3.1 Mesh Input Specification with XML

Our implementation starts with the design of our DSL, which contains the mesh-related information described in the commonality analysis (Appendix A). As mentioned in Chapter 2, the primary structure of our DSL is a property list. The design of our DSL follows two principles. First, the DSL should be declarative, regardless of whether it is embedded in a GPL or stand-alone. The word "declarative" means that the DSL for building mesh generators should only describe what the requirements for the mesh generator are, such as the boundary description and output file formats. The DSL should not imply how a mesh generator or a part of the mesh generator is built, which is the job of the program generator. Second, the nature of the mesh generator application implies that the DSL should have a top-down structure. This top-down structure is also visible by inspecting and studying the relationship between the set of input data required to generate a mesh generator. For instance, a mesh application in our research scope requires input in three major categories: geometry description, physical attributes, and output file specification. Each category can be further divided in detail until the input can be specified by a single value. For example, the geometry description of the domain can be summarized by the specification of different zones, where each zone is further specified by the coordinates of each boundary vertex. As another example, the physical attributes of a mesh application can start with the set of material specifications, where each material is described by a set of material properties, each of which contains a value. The top-down design principle of our DSL is not only logical, it is also capable of reducing input errors, since each category of input information (e.g. geometry, physical attributes, etc.) is specified in an orderly fashion. A specific category should be fully expanded until each piece of input is provided with a single value.

At the initial stage of developing our DSL to model mesh applications, we have two options: we can either develop a stand-alone DSL with a new syntax and compiler, or we can develop a DSL embedded in an existing programming language. After careful consideration, we decided to go with the second idea because the extra effort to develop a language parser to fit a new DSL is unnecessary. An abundance of research literature [23, 25, 30] has shown that it is not only easy to use an existing language to implement a DSL, but it is also convenient to develop a language translator since many languages offer APIs and tools that provide the parsing capabilities [6, 34, 35]. During our research, we have carefully considered two types of languages as design options. They are a functional programming language or Extensible Markup Language (XML).

Functional Languages with Domain-specific Embedding

Functional languages with functional composition and higher-order typing offer many capabilities that can be used to create a domain-specific language [23, 44]. Functional composition, overloading, monads and data types are tools to create an abstraction that can serve as a DSL.

One example of embedding domain-specific functionality within functional language is wxHaskell [32], which provides an interface to the wxWidgets [55] GUI library. The WX library part of the implementation applies functional abstractions to the user interface components. This allows Haskell programs to create user interfaces and it allows further programming logic to easily connect to the user interface components and events. "Most of the hierarchical structure of user interface components is lost in the functional abstractions" [14].

The use of functional languages was explored as a possible host language to model mesh information at the early stage of our DSL design. However, the distinctive functional abstraction capability offered by a functional language is not needed for the declarative data required to generate a mesh generator. Also, the top-down hierarchical structure can be naturally implemented in languages such as XML.

Use of XML to Encapsulate Data in Markups

XML is a mark-up language used to provide structured information. The application of XML in DSL design has been explored by others. Examples of using XML syntax for DSLs include MathML [11], MatML [22], etc. One of the advantages of this practice is that XML provides flexible parsing. Basic XML parsing techniques such as tree-parsing and event-parsing have been implemented in many APIs and frameworks [35].

We decided to use XML as the implementation language for our DSL. First of all, XML is designed to encapsulate structured information. The nature of nested markup elements in XML can be used to reflect a hierarchical relationship between mesh related inputs. For example, a mesh contains elements, an element contains vertices, etc. Secondly, the existing parsing and binding technologies such as [35] offer convenient tools to process the data in an XML document. Thirdly, the examples shown in a similar field in MatML [22] strengthen our belief that XML works for our purpose. MatML [22] is an application of XML to model material data, aimed at improving the interpretation and interoperability of communicating material property data. Since our DSL contains material information, the practice of XML in material data can be expanded to describe other types of information.

The document structure in XML can be defined in two ways: DTD (Document Type Definition) or Schema. Figure 3.1 and 3.2 provide fragments of a DSL implementation that describes the mesh element and material specifications, respectively.

Figure 3.1 is a sample specification of a set of elements in terms of node locations. The $< elementSet >$ is the top element, which contains subelements $< geometrySpec >$, $< nodeGeo >$, $< node >$, $< location >$. In this specification, we are describing the triangular elements that the mesh generator will generate. Each element consists of three geometric nodes located at the three triangular vertices. The hierarchical structure ensures that the top element must be closed after all the subelements have been properly closed. The location of a node within an element is specified using the natural coordinate system.

```
<elementSet>
    <geometrySpec>
        <shape>triangle</shape>
        <nodeGeo count=``3">
            <node id="1">
                <location>1,0,0</location>
            </node>
            <node id=``2">
                <location>0,1,0</location>
            </node>
            <node id=``3">
                <location>0,0,1</location>
            </node>
        </nodeGeo>
    </geometrySpec>
</elementSet>
```

Figure 3.1: DSL Fragment for Element Specification

The natural coordinates for a triangle are shown in Figure 3.2, where the coordinates of point P are as follows:

$$P = (L_1, L_2, L_3) = (\frac{A_1}{A}, \frac{A_2}{A}, \frac{A_3}{A})$$

As an example, if point P is at the centroid of a triangle of area A, then $A_1 = A_2 = A_3 = \frac{A}{3}$. Therefore, the natural coordinates of point P are $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

Figure 3.3 is a sample specification of the material information required to produce a mesh generator. The $< materialSet >$ is the top element, which contains subelements $< material >$, $< id >$, $< matproperty >$, $< name >$, $< type >$, $< value >$. In this specification, there is one material with one material property, which is Poisson's ratio with a value of 0.3. A more complex material specification can be written by adding more material properties and materials. Before adding a new material in the specification, the

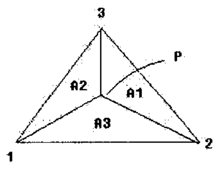


Figure 3.2: Natural Coordinate System of a Triangular Element

last < *material* > must be properly closed.

```
<materialSet>
    <material>
        <id> 1 </id>
        <matproperty>
            <name> Poisson ratio </name>
            <type> real </type>
            <value> 0.3 </value>
        </matproperty>
    </material>
</materialSet>
```

Figure 3.3: DSL Fragment for Material Specification

3.2 Meshing Algorithm

The mesh generation process for all mesh generators in the program family is handled by the same meshing algorithm, which was decided at scope time. For our research, we adopt the isoparametric meshing scheme proposed by Zienkiewicz and Phillips [56]. We chose this semi-automated algorithm because it is designed for structured mesh generation and because it accommodates a wide variety of spatial configurations. The essence of this algorithm, which we will call the ZP algorithm, is the use of isoparametric curvilinear mapping of quadrilaterals, which allows a unique coordinate mapping of curvilinear and Cartesian coordinates. For a simple rectangular domain in which the Cartesian coordinates of the four corner nodes are specified, the coordinate for any node in the domain can be calculated using the four shape functions associated with the corner nodes. A shape function in the ZP algorithm is used to interpolate the geometry by defining the mapping between curvilinear and Cartesian coordinates.

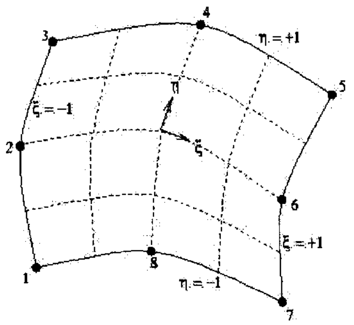


Figure 3.4: A Parabolic Quadrilateral Mesh Calculated with Eight-node Shape Functions

A simple example of curvilinear coordinate system in a quadrilateral mesh is shown in Figure 3.4. If the boundary contains curved edges, a set of eight-node shape functions may be necessary to calculate the coordinates of nodes in the mesh. Given the coordinates of the corner vertices and possibly midpoints of each boundary edge, any point P with coordinate x, y, and z within the domain can be calculated using the shape functions on curvilinear coordinates. For example, any point P in the domain is calculated by the Cartesian coordinates of the eight boundary nodes (x_i, y_i, and z_i with $1 \leq i \leq 8$) and the associated shape functions. The calculations of the coordinates of point P are done as follows:

$$x = \sum_{i=1}^{8} N_i x_i \quad y = \sum_{i=1}^{8} N_i y_i \quad z = \sum_{i=1}^{8} N_i z_i \tag{3.1}$$

where N_i is a shape function associated with node i. Each N_i is defined in terms of a curvilinear coordinate system ξ and η, which has values ranging from -1 to 1 on opposite sides of the quadrilateral shaped domain. For an eight-node element, the shape functions are listed as follows [56]:

$$\begin{aligned} N_1 &= -\frac{1}{4}(1-\xi)(1-\eta)(\xi+\eta+1), & N_2 &= \frac{1}{2}(1-\xi)(1-\eta^2) \\ N_3 &= \frac{1}{4}(1-\xi)(1+\eta)(-\xi+\eta-1), & N_4 &= \frac{1}{2}(1-\xi^2)(1+\eta) \\ N_5 &= \frac{1}{4}(1+\xi)(1+\eta)(\xi+\eta-1), & N_6 &= \frac{1}{2}(1+\xi)(1-\eta^2) \\ N_7 &= \frac{1}{4}(1+\xi)(1-\eta)(\xi-\eta-1), & N_8 &= \frac{1}{2}(1-\xi^2)(1-\eta) \end{aligned} \tag{3.2}$$

In the current study, we are assuming that the domain boundaries have straight edges. Therefore, the implementation for the current study will use 4-noded elements and their associated shape functions.

The ZP algorithm can be extended to a general scheme that applies for arbitrary domain shapes. To do this, the ZP algorithm uses the curvilinear mapping from a key diagram to the actual geometric domain. An example key diagram used for an arbitrary shape is shown in Figure 3.5. The key diagram is a chequerboard pattern of quadrilaterals, which we refer to as “super” elements. Each “super” element uses the curvilinear system in the same way as used for one element in Figure 3.4. The ZP algorithm requires three types of data inputs to define the domain: span definitions, specified points, and zone specification. The span definitions are the number of rows and columns in the key diagram. The number of "super" elements is the product of the number of rows and columns. The specified points are the Cartesian coordinates of the corner vertices in each "super" element. The zone definitions define the material assignment for different parts of the domain. Geometrically, a zone corresponds to one "super" element or multiple "super" elements combined. Each zone is assigned by a material and the set of material properties. If such a material is not given, the corresponding zone is considered as void. Void zones are useful to define multiply-connected domains. For example, in Figure 3.5, zone 4 and 6 are void zones and they represent the hole in the bottom part of the actual domain.

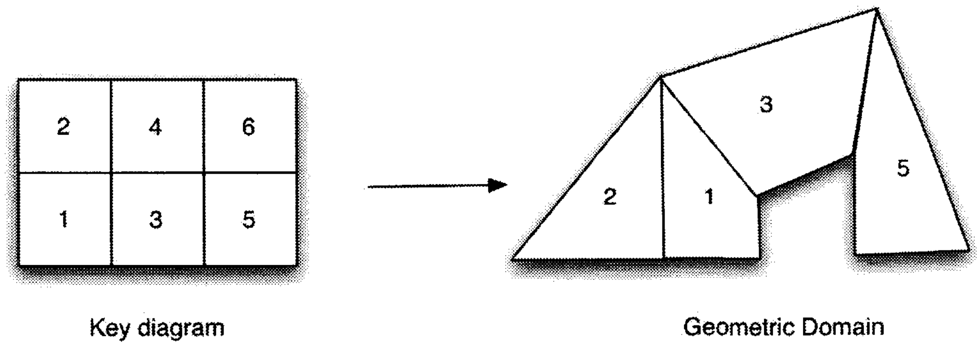


Figure 3.5: General Scheme of ZP Algorithm with Key Diagram Mapping

To generate a structured mesh with the ZP algorithm, the number of equal subdivisions for each row and column must be specified. The nodal numbering are generated and traversed sequentially across a specified direction. The direction is either horizontal if the number of column subdivisions is greater than the number of row subdivisions or vertical if this is not the case. A "super" element in the key diagram uses the subdivision numbers to create Cartesian coordinates using shape functions in the manner shown by Equation 3.1. Our implementation of the ZP algorithm written in Java is provided in Figure 3.6. After all nodal points are generated, the mesh elements are produced by connecting nodal points according to element connectivity, which follows a pre-determined pattern depending on the required topology pattern and the shape of the elements.

```
public class meshGeneration

{

/* Variables Declarations. This section contains the input variables and other variables that

will be used to store the results */


/* All the input variables are declared as follows. The values of these input variables have been

initialized. Please note that the super element numbering is always vertically upwards.  The r and s direction

refer to the horizontal and vertical directions in the curvilinear system in Figure 3.4.*/

  int nrows;                // the number of rows in the key diagram

  int ncols;                // the number of columns in the key diagram

  int [] nelr;              // the number of subdivisions in the r-direction in each row, initialized to 0 for all entries

  int [] nels;              // the number of subdivisions in the s-direction in each column,initialized to 0

  boolean [] zone_valid;    // the array to identify void zones, a zone is 1 if void and 0 otherwise

  float [] x_super;         // the x coordinates of all super elements

  float [] y_super;         // the y coordinates of all super elements


/* All the other variables that store the results are delcared here */

float [] x_coord, y_coord;          // the arrays to hold the x, y coordinates of the nodes in the mesh

float [] r_coord, s_coord;          // the arrays to hold the r, s coordinates of the nodes in the mesh

int nnodes;                         // the number of nodes in the mesh

int nzones:                    // the number of zones in the mesh

int r_sub, s_sub;                   // the total number of subdivisions along r and s directions

int [] zone_id;                // zone id is stored for each node
/* the method that implements the meshing algorithm is as follows. Please note that this piece
of code only deals with row based nodal numbering */

   public void generate()
   {
        for(int i = 0; i < nelr.length; i++)
        {
           r_sub += nelr[i];     // calculate the value of r_sub
        }
        for(int i = 0; i < nels.length; i++)
        {
           s_sub += nels[i];     // calculate the value of s_sub
        }
        nzones = nrows * ncols                // calculate the number of superelements in the key diagram
        nnodes = (r_sub + 1) * (s_sub + 1);  // calculate the number of nodes in the mesh
        r_coord = new float[nnodes];     // initialize values of r_coord
        s_coord = new float[nnodes];     // initialize values of s_coord
        x_coord = new float[nnodes];     // initialize values of x_coord
        y_coord = new float[nnodes];     // initialize values of y_coord
        zone_id = new int[nnodes];       // initialize zone_id array
        int nodeindex = 0;              // initialize the counter to increment the nodal numbering
        float r,s;                      // local variable to hold r and s coordinates for a node
        for(int row_index = 1; (row_index < nrows + 1); row_index++)
        {
                int k = row_index;                    //  the variable k keeps track of the change of superelements
                for(int i = k; (i <= nelr[((k-1) % nrows)] + 1); i++)           // loop over (k-1)%nrows times
                {
                        s = (float)(-1 + 2*(i-1)/(float)nelr[((k-1) % nrows)]);  // calculate s
                        if(k < nzones - 1)              // if the superelement is not at the rightmost position
                        {
                                for(int j = 1; j<= nels[Math.ceiling(k/nrows) - 1] + zone_valid[k+nrows-1]; j++)
                                // loop over rows in r-s space
                                {
                                        r = (float)(-1 + 2*(j-1)/(float) nels[Math.ceiling(k/nrows) - 1]); // calculate r
                                        r_coord[nodeindex] = r;                                  // set r_coord value
                                        s_coord[nodeindex] = s;                                 // set s_coord value
                                        zone_id[nodeindex] = k;                                  // set zone_id array value
                                        nodeindex++;                                             // increment node counter
                                }                                                                //end j for loop
                                k = k + nrows;                                          // go to the next row

                        }
                        else
                        {                                 // if not, then the superelement is at the rightmost position
                                for(int j = 1; j<= nels[Math.ceiling(k/nrows) - 1]; j++) // loop over rows in r-s space
                                {
                                        r = (float)(-1 + 2*(j-1)/(float)(nels[Math.ceiling(k/nrows) - 1])); //calculate r
                                        r_coord[nodeindex] = r;                                  //set r_coord value
                                        s_coord[nodeindex] = s;                                  //set s_coord value
                                        zone_id[nodeindex] = k;                                  // set zone_id array value
                                        nodeindex++;                                             // increment node counter
                                }                                                       //end j for loop
                                k = row_index;                              // go to the first superelement in that row
                        } //end else
                } //end i loop
         } //end row_index loop
        /* Now Calculate the Cartesian coordinates using four-node shape functions */
        for(int i = 0; i < nnodes; i++)    // go over all nodes
        {
                int k = zone_id[i];          // find the superelement the node i is in
                r = r_coord[i];              // get r value for node i
                s = s_coord[i];              // get s value for node i
                // calculate x coordinate for node i using shape function
                x_coord[i] = x_super[k-1][0] * ((1-r)*(1-s)/4) + x_super[k-1][1]*((1+r)*(1-s)/4) + x_super[k
-1][2]*((1+r)*(1+s)/4) + x_super[k-1][3] *((1-r)*(1+s)/4);
                // calculate y coordinate for node i using shape function
                y_coord[i] = y_super[k-1][0] * ((1-r)*(1-s)/4) + y_super[k-1][1]*((1+r)*(1-s)/4) + y_super[k
-1][2]*((1+r)*(1+s)/4) + y_super[k-1][3] *((1-r)*(1+s)/4);
          } // end loop. now all the nodes have been created
     }//end generate method
}//end class
```

3.3 Formatting Output Files with XSLT Stylesheets

One of the major contributions of our research is to design mesh generators that allow customization of output file formats. To do this, we use XSLT (Extensible Stylesheet Language Transformation) [12] to organize mesh data in different file(s). There are two steps involved in the output file production process. First, after the mesh is produced, all the relevant data that may be useful in a finite element program will be stored in a separate XML document in a pre-defined structure. Second, according to the structure definition of the XML document that contains the mesh data, the XSL stylesheet file(s) must be created to specify the output file formats. The XSL files are then specified either at specification time or at run time. If the stylesheet is parsed without errors, the transformation process is performed automatically, and the output files will be created as indicated by the stylesheet.

This section is organized as follows. Section 3.3.1 gives the essentials of the XML document structure that contains the mesh data after the mesh has been produced. Section 3.3.2 introduces the use of XSLT in general and explain how the stylesheet functionality is integrated into our system to produce plain text files.

3.3.1 The Structure of XML Document Containing the Mesh Data

The mesh generation process is followed by a function to organize mesh data according to the pre-defined XML document structure. This function uses basic file input/output handling to open a new file, wrap data in pre-defined tags, and close the file when the writing process is complete. The resulting XML document will be stored in the directory in which the system files are located. Although this XML document can be directly opened to view the contents, this action is not encouraged. Instead, the document definition documents of the XML file will be presented so that different stylesheets can be written to indicate how the mesh data should be organized in the output files. Figure 3.5 is the DTD (Document

Type Definition) of the XML document holding mesh data information.

A DTD file defines the structure of an XML document by listing the parent child relationship between elements. A DTD document uses two symbols: $<!ELEMENT>$ and $<!ATTLIST>$. A $<!ELEMENT>$ defines an element by its name and the set of child element between a pair of brackets. The top element in a DTD is the root element in the XML document structure. If an element has child elements, the child elements are defined in the same order as they appear in the child element list. For example, in the structure of mesh data file, the root element is $< MeshData >$. The root element has six child elements, each of which defines a specific category of information that may be used by finite element programs. An $<!ATTLIST>$ defines attributes embedded in an element. An attribute is either marked by #REQUIRED if they must be assigned a value or #IMPLIED if they can be omitted. In a DTD document, the element list is usually created first, followed by the attribute list. A fragment of a sample XML document created by this DTD is shown in Figure 3.6. In this example, we assume that the number of degrees of freedom for each node is 1. Therefore, the fixity value is either 0 if it is fixed or 1 if it is free. In more complex examples, a fixity may be a sequence of numbers, with each element in the sequence associated with the corresponding sequence of a degrees of freedom.

Figure 3.6 contains a small fragment of XML document holding geometric data of a mesh. The $< VertexSet >$ iterates over all vertices in a mesh and organize them based on x, y, and z coordinates. Fixity is also included in $< VertexSet >$, which is a list of zero or one values. The set of elements is represented by $< ElementSet >$, which has three required attributes. Each element defined in $< ElementSet >$ must be given an ID, its vertex coordinates, and the locations of nodes for the degrees of freedom. The reason that the vertex coordinates are stored twice is that the structure is aimed at providing maximum flexibility in producing file outputs. The set of vertex information is usually required by finite element programs, either separately or embedded in element data. Although it may

```
<!ELEMENT MeshData (domainInfo,VertexSet,ElementSet,ZoneSet,
BoundaryConditionSet,SysParameterSet)>
<!ELEMENT domainInfo(author,title,comment)>
<!ELEMENT VertexSet(x_coord,y_coord,z_coord,fixity)>
<!ELEMENT ElementSet (Element)>
<!ELEMENT Element (ElementGeo,ElementDof)>
<!ELEMENT ElementGeo (Vertex)>
<!ELEMENT Vertex (x_coord,y_coord,z_coord,fixity)>
<!ELEMENT ElementDof (Node)>
<!ELEMENT Node (x_coord,y_coord,z_coord,dofName)>
<!ELEMENT ZoneSet (zone)>
<!ELEMENT zone (elements,material)>
<!ELEMENT elements (elementid)>
<!ELEMENT material (matproperty)>
<!ELEMENT matproperty (name,type,value)>
<!ELEMENT BoundaryConditionSet (BoundaryCondition)>
<!ELEMENT BoundaryCondition (name,type,location,value)>
<!ELEMENT SysParameterSet (SysParameter)>
<!ELEMENT SysParameter (name,value)>
<!ATTLIST ElementSet shape CDATA #REQUIRED>
<!ATTLIST ElementSet NumOfVertices CDATA #REQUIRED>
<!ATTLIST ElementSet NumOfNodes CDATA #REQUIRED>
<!ATTLIST ElementSet LocalNumbering CDATA #REQUIRED>
<!ATTLIST Element id CDATA #REQUIRED>
<!ATTLIST Vertex id CDATA #REQUIRED>
<!ATTLIST Node NumOfDof CDATA #REQUIRED>
<!ATTLIST ZoneSet NumOfZones CDATA #REQUIRED>
<!ATTLIST zone id CDATA #REQUIRED>
<!ATTLIST material id CDATA #REQUIRED>
<!ATTLIST material name CDATA #REQUIRED>
<!ATTLIST material type CDATA #REQUIRED>
```

Figure 3.6: DTD of XML Document Holding Mesh Data

```
<MeshData>
  <ElementSet shape="triangle" NumOfVertices="3" NumOfNodes="3"
     LocalNumbering="1 2 3">
        <Element id="1">
            <ElementGeo>
                <Vertex id="1">
                    <x_coord> 5 </x_coord>
                    <y_coord> 3 </y_coord>
                    <z_coord> 0 </z_coord>
                    <fixity> 1 </fixity>
                </Vertex>
                <Vertex id="2">
                    <x_coord> 2 </x_coord>
                    <y_coord> 3.5 </y_coord>
                    <z_coord> 0 </z_coord>
                    <fixity> 1 </fixity>
                </Vertex>
                 <Vertex id="3">
                    <x_coord> 2.5 </x_coord>
                    <y_coord> 3 </y_coord>
                    <z_coord> 0 </z_coord>
                    <fixity> 1 </fixity>
                </Vertex>
            </ElementGeo>
            <ElementDof>
                <Node NumOfDof="1">
                     <x_coord> 5 </x_coord>
                    <y_coord> 3 </y_coord>
                    <z_coord> 0 </z_coord>
                    <dofName> x displacement </dofName>
                </Node>
            </ElementDof>
        </Element>
    </ElementSet>
</MeshData>
```

Figure 3.7: A Fragment of XML Document Defining Mesh Data

be converted between different formats by using stylesheets, the extra work required by stylesheets is more difficult than writing the same information twice in the XML document. Therefore, we choose to introduce the structure as shown in Figure 3.5.

3.3.2 Using XSLT Stylesheets to Produce Plain Text Files

XSLT [12] is a domain-specific markup language aimed at XML document transformation and presentation. XML is a language that describes data. However, if the data needs to be transformed for web display or other processing requirements, XSLT is often used. The stylesheet functionality provided by XSLT is powerful in that it can convert between multiple types of documents, such as XML to XML, XML to HTML, XML to plain text files, etc. Therefore, when we made the decision to use XML to model mesh data. XSLT became a good choice to meet our requirement of file transformation processing.

In the age of the Internet, formats such as HTML clearly dominate the application of XSLT on the output side. However, plain text is usually required by finite element programs. The transformation from XML to text is done by specifying the method of the output format $< xsl : output >$ as text. However, we need to choose an XSLT processor as an environment for transformation process. The XSLT processor provides a run-time command to create text files through transformation, as well as extensions and tools for advanced functions ($e.g.$ apply multiple stylesheets to a single XML data source). The XSLT processor we adopted to implement our system is Xalan [43].

The stylesheet files written based on the XML structure introduced in Figure 3.5 should be prepared by the users of our system. Figure 3.7 is an example XSL stylesheet to produce a text file, which contains the coordinate information for the vertices in the first element. The transformation inserts a tab between each coordinate value. In the example shown in Figure 3.7, the XSL stylesheet uses $< xsl : template >$ and $< xsl : value{-}of >$ elements

```
<?xml version="1.0"?> <xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output method="text"/>
<xsl:variable name="newline"><xsl:text>
</xsl:text></xsl:variable>
<xsl:variable name="tab"><xsl:text>&#x09;</xsl:text></xsl:variable>
<xsl:template match="MeshData">
<xsl:text>X Coordinate&#x09;Y Coordinates&#x09;
Z Coordinates&#x09;Fixity</xsl:text>
<<xsl:value-of select="$newline"/>
<xsl:apply-templates select="Vertex"/>
</xsl:template>
<xsl:template match="Vertex">
<xsl:value-of select="x_coord"/><xsl:value-of select="$tab"/>
<xsl:value-of select="y_coord"/><xsl:value-of select="$tab"/>
<xsl:value-of select="z_coord"/><xsl:value-of select="$tab"/>
<xsl:value-of select="fixity"/><xsl:value-of select="$newline"/>
</xsl:template>
</xsl:stylesheet>
```

Figure 3.8: Sample XSL Stylesheet File

to apply templates and extract values between XML element tags. XSL stylesheets also offers programming facilities for further data manipulation. For example,$< xsl : variable >$ is used to declare special symbols for later use. XSLT embeds advanced string-handling functionality, such as testing string patterns, adding or removing characters from a string, etc.. However, as mentioned earlier, the use of the advanced XSLT functionality is up to the user of our system, because they are responsible for preparing and validating the stylesheets that they need. The resulting text file is shown in Figure 3.8.

```
X Coordinate    Y Coordinate    Z Coordinate    Fixity
5                3                  0           1
2                3.5                0           1
2.5              3                  0           0
```

Figure 3.9: Output Text File

Chapter 4

System Demonstration

This chapter provides examples of a proof of concept PMG - Parameterized Mesh Generator. As mentioned in previous chapters, the implementation of the system should reflect the distinction between program family members. For example, fixing the parameters of variation to different values should produce different mesh generators. Binding the value of a variability at specification time or run time is considered to have the same effect. The examples provided in this chapter focus on the variabilities and show that changes made to parameters of variation at specification time lead to changes in the resulting mesh generator.

The chapter includes two example spatial domains discussed in separate sections. The first example is meshing a rectangular domain with four zones. The mesh generator used in this example is generated from an empty specification and therefore is the most general-purpose mesh generator in the family. The second example involves an irregular domain with six zones, two of which are void zones.

4.1 General-purpose Interface for Rectangular Domain Meshing

As mentioned in Chapter 3, we use a meshing algorithm which adopts the mapping between curvilinear and Cartesian coordinates. The curvilinear mapping is used in the key diagram where the "super" element is defined. In this example, the key diagram is of the same geometric shape as the actual geometric domain. Figure 4.1 illustrates the key diagram.

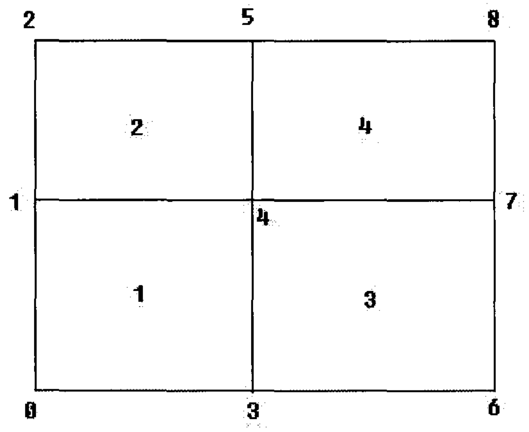


Figure 4.1: Key Diagram of Quadrilateral Domain

We use the most general-purpose mesh generator in the family for this problem. The specification corresponding to the general-purpose family member is described in Figure 4.2. It only contains the root element of the data definition.

```
<?xml version="1.0"?>
<MeshGeneratorApplication>
</MeshGeneratorApplication>
```

Figure 4.2: XML Specification for General-purpose Mesh Generator

The resulting mesh generator is produced with interface shown in Figure 4.3. The general-purpose mesh generator allows all the seed information to be entered at run time. Therefore, the values of all the inputs shown in dialog boxes are blank. To illustrate each menu options, we will include the screen shots of the system interface in sections 4.1.1-4.1.6.

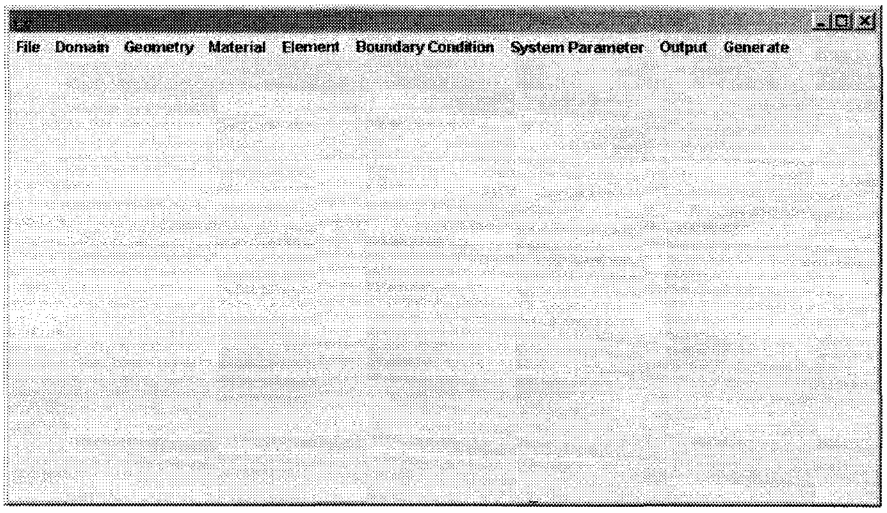


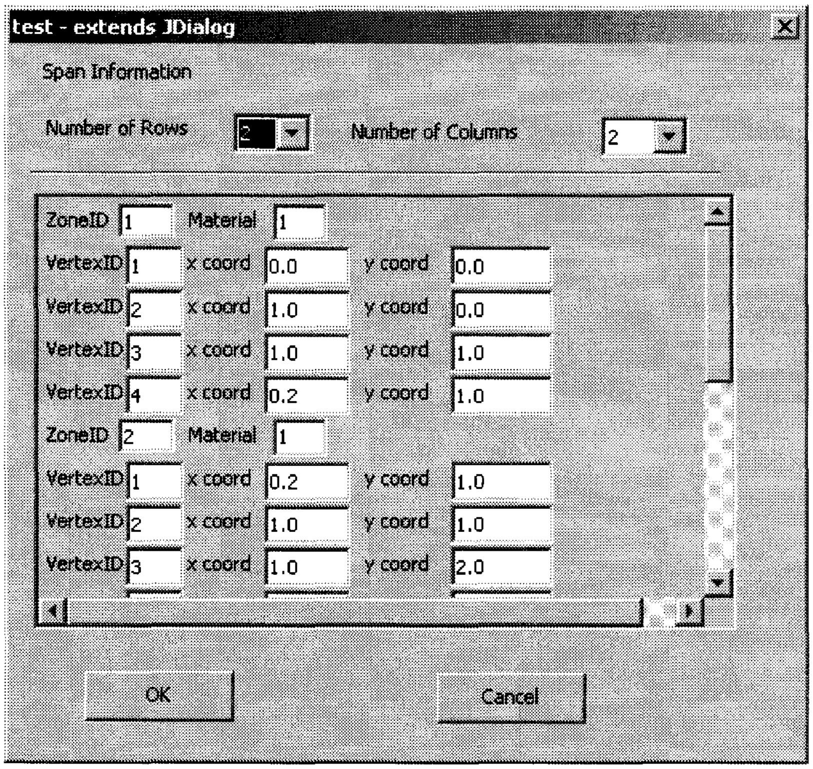
Figure 4.3: GUI for a Mesh Generator

4.1.1 File and Domain Menus

The first two menus are "File" and "Domain". The "File" menu has three options: Open, Save, and Exit. The user clicks on "Open" to open an existing file of a mesh, "Save" to save the current mesh, and "Exit" to exit the system. The "Domain" menu allows the user to enter the domain information, which consist of the following four types of information: domain, author, date, and comments. Domain stands for the application domain for which the mesh is intended (*e.g.* mechanic, thermodynamic, *etc*). The information can be directly written to the output file(s).

4.1.2 Geometry Dialog

The geometry dialog allows the user to enter the geometric information, such as the co-ordinates of the vertices in the key diagram. The key diagram contains a span of "super" elements, where the user must specify the number of rows and columns. After that, the user must provide the coordinates for each zone, as well as the assigned material identifier. Since our scope only considers boundaries with straight edges, each zone is set to have four vertices. Figure 4.4 displays the geometry dialog with the geometric information associated with this example.



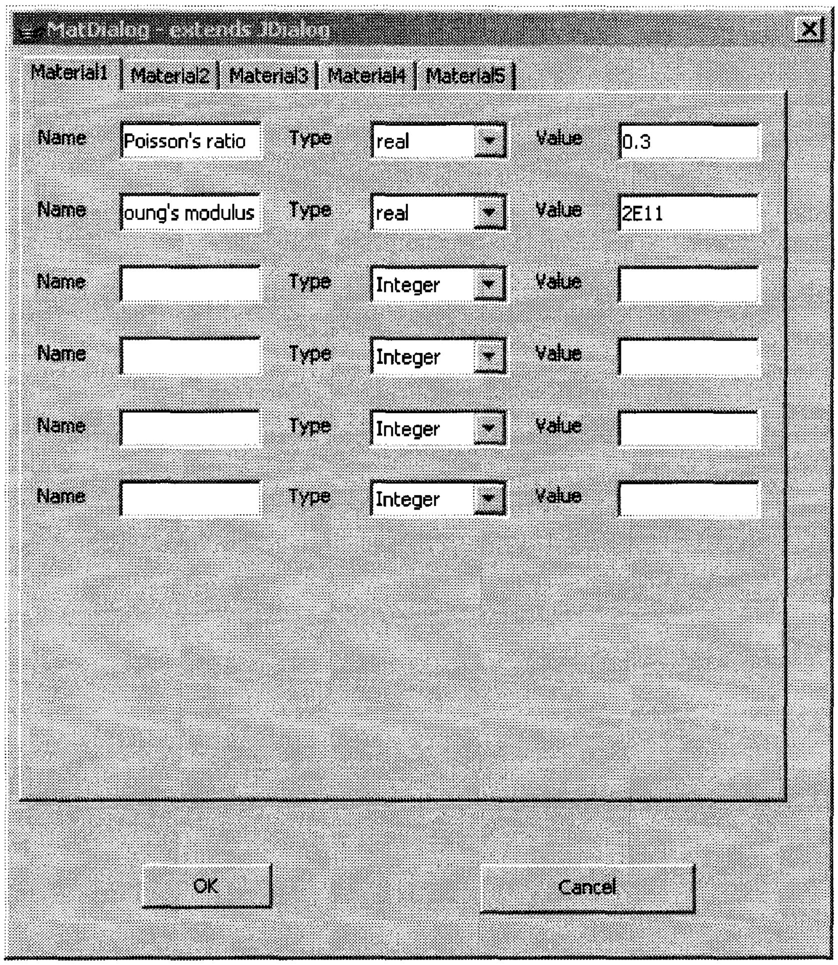
Number of Rows 2 Number of Columns 2

ZoneID 1 Material 1
VertexID 1 x coord 0.0 y coord 0.0
VertexID 2 x coord 1.0 y coord 0.0
VertexID 3 x coord 1.0 y coord 1.0
VertexID 4 x coord 0.2 y coord 1.0
ZoneID 2 Material 1
VertexID 1 x coord 0.2 y coord 1.0
VertexID 2 x coord 1.0 y coord 1.0
VertexID 3 x coord 1.0 y coord 2.0

Figure 4.4: Specified Information in Geometry Dialog

4.1.3 Material Dialog

The material dialog allows the user to enter the material information associated with the problem. The general-purpose system allows a maximum of five materials to be specified, each of which can have up to five different material properties. Each material property is described by a name, type, and value. An example of material dialog with the specified information is shown in Figure 4.5.



MatDialog - extends JDialog

Material1 | Material2 | Material3 | Material4 | Material5

Name	Poisson's ratio	Type	real	Value	0.3
Name	oung's modulus	Type	real	Value	2E11
Name		Type	Integer	Value	
Name		Type	Integer	Value	
Name		Type	Integer	Value	
Name		Type	Integer	Value	

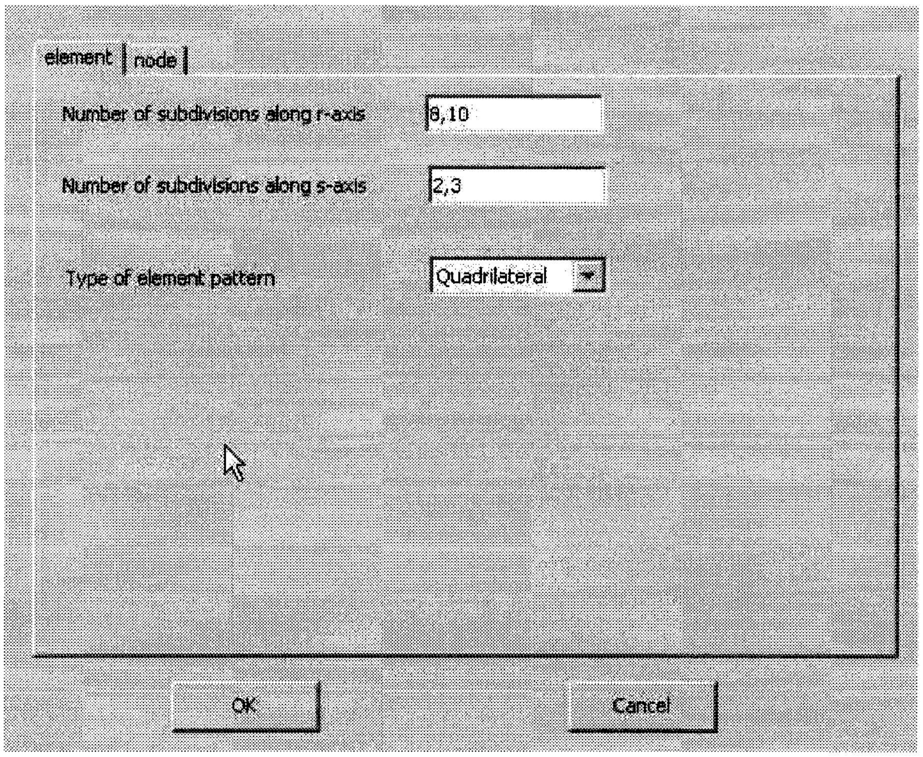
Figure 4.5: Specified Information in the Material Dialog

4.1.4 Element Dialog

The element dialog allows the user to enter information associated with the mesh elements. The dialog has two parts: subdivision and node. In the first part, the number of subdivisions for each row and column along each direction must be specified. Subdivision values for each row and column are separated by commas. The shape of the element must also be specified. The possible topology patterns to be selected are quadrilaterals and eight triangular patterns, corresponding to the parameters of variation documented in the commonality analysis(Appendix A). The node information is related to the nodes that are used to interpolate the geometry or that are assigned with degrees of freedom. The location of a node in this case is in the local coordinate system used by the shape functions in Chapter 3. For nodes used for degrees of freedom, the names of degrees of freedom on each node can also be specified. If there are multiple degrees of freedom, their names should separated by commas. In Figure 4.7, the two degrees of freedom are "x_displacement" and "y_displacement". Figure 4.6 and 4.7 illustrate the two parts of the dialog with the user specified information for the current problem.

4.1.5 Boundary Condition Dialog

The boundary condition dialog allows the user to enter boundary conditions associated with the problem. In this example, the boundary conditions in terms of fixity and traction are specified as in Figure 4.8. The boundary where each boundary condition is applied is a pair of vertices separated by commas in the key diagram. The handling of boundary conditions in the prototype of MGG assumes that the entire edge has the same boundary condition.



Number of subdivisions along r-axis 8,10

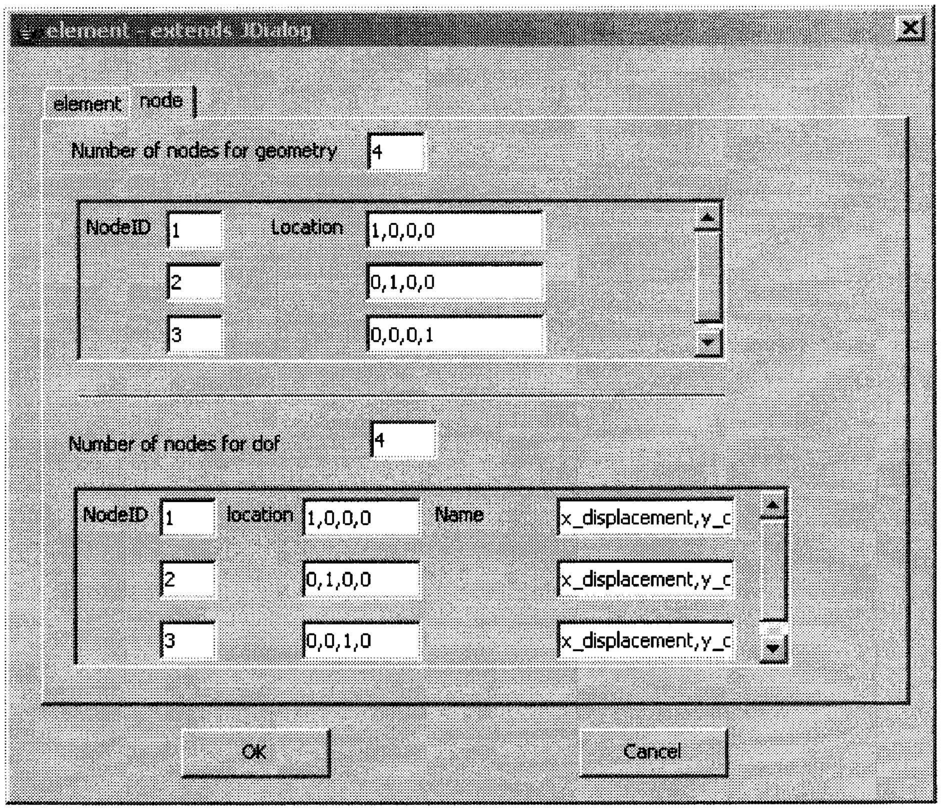
Number of subdivisions along s-axis 2,3

Type of element pattern Quadrilateral

Figure 4.6: Specified Subdivisions in the Element Dialog

4.1.6 System Parameter Dialog

The system parameter dialog lets the user enter any system-related information, which is directly written to output file(s). Examples of system parameters are degree of implicitness for a time marching algorithm, spring stiffness for artificially constraining degree of freedom, the maximum number of iteration, etc. The prototype of MGG allows a maximum of three systems parameters to be specified. Figure 4.9 is an example of a system parameter dialog with user specified information.


element - extends JDialog

element | node

Number of nodes for geometry: 4

NodeID	Location
1	1,0,0,0
2	0,1,0,0
3	0,0,0,1

Number of nodes for dof: 4

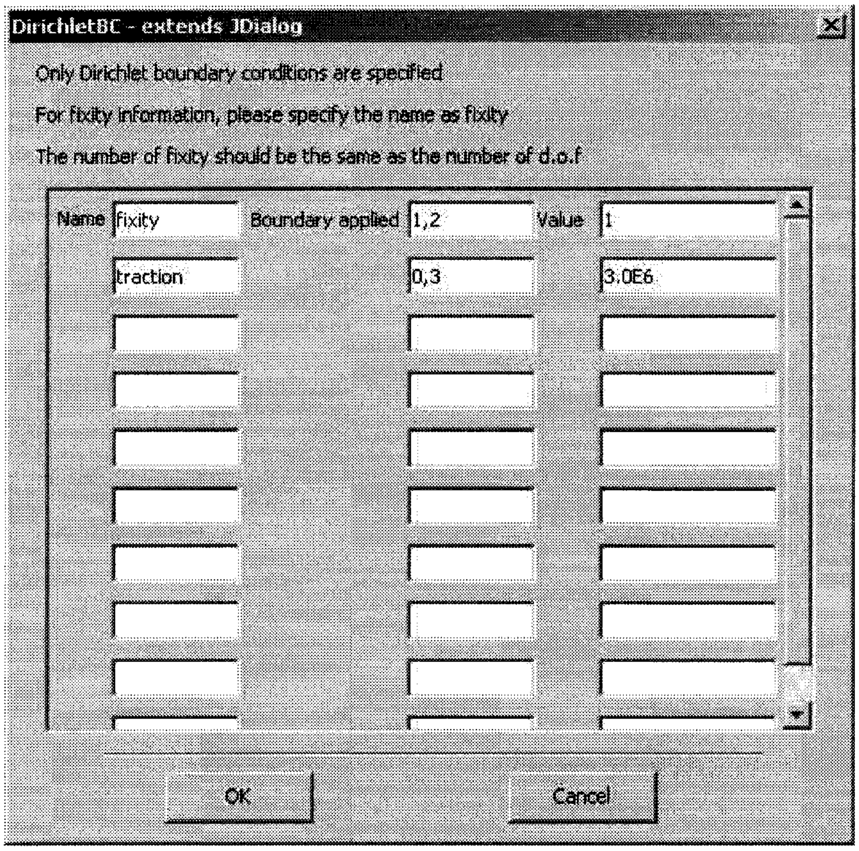
NodeID	location	Name
1	1,0,0,0	x_displacement,y_c
2	0,1,0,0	x_displacement,y_c
3	0,0,1,0	x_displacement,y_c

Figure 4.7: Specified Node Information in Element Dialog

4.1.7 Generating the Mesh and Visualization

After the input information is specified, the mesh can be generated by clicking “Generate”. The resulting quadrilateral mesh of our example is shown in Figure 4.10.

Since this is the general-purpose mesh generator, we can change some values and see the changes in the resulting mesh. One common variability is to change the shape of the element. The interface allows the user to specify six patterns of triangular meshes. Changing the value in the element dialog to “Triangle6” results in the mesh shown in Figure 4.11.



Only Dirichlet boundary conditions are specified
For fixity information, please specify the name as fixity
The number of fixity should be the same as the number of d.o.f

Name	Boundary applied	Value
fixity	1,2	1
traction	0,3	3.0E6

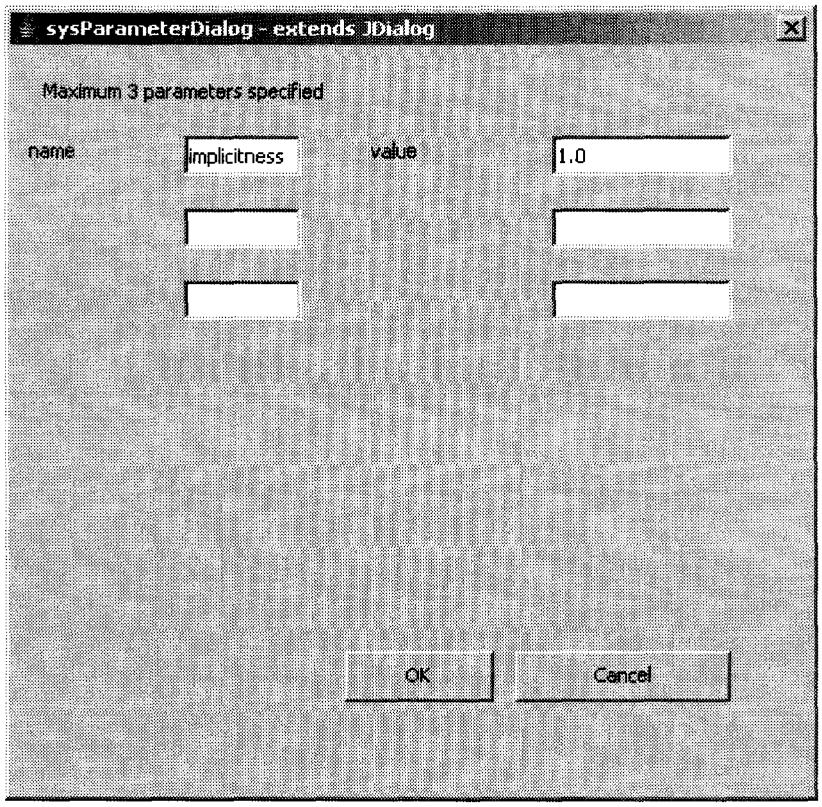
Figure 4.8: Specified Information in the Boundary Condition Dialog

4.1.8 XSL Stylesheet and Output File Generation

In the current example, we specified two stylesheet files for file generation. The contents of the files are shown in Figure 4.12 and Figure 4.13. The first stylesheet contains instructions to print only vertex coordinate information, while the second stylesheet is to print only element connectivity information.

The output file dialog allows the user to specify the XSL stylesheet file to be used for customizing output file(s). Figure 4.14 is a screen shot of output file dialog with specified XSL file locations for the quadrilateral mesh shown in Figure 4.10.

The resulting text files are shown in Figure 4.15 and 4.16. Since the mesh has too many


sysParameterDialog - extends JDialog

Maximum 3 parameters specified

name implicitness value 1.0

Figure 4.9: Specified Information in System Parameter Dialog

elements, we will only print a portion of contents of the output files.

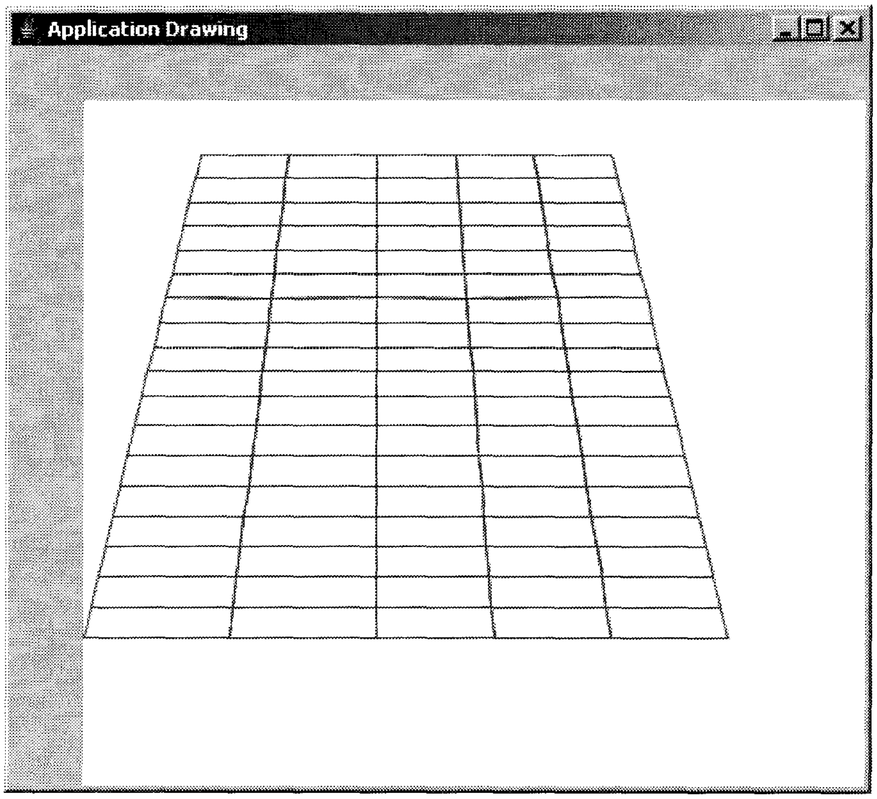


Figure 4.10: Generated Quad Mesh

4.2 Special-purpose Interface for a Nonrectangular Domain

The first example shows a simple rectangular domain using the most general-purpose mesh generator in the family. In this example we will show how a special-purpose mesh generator is produced with input values bound at specification time in the XML specification, and how this mesh generator program differs from the first one.

Before we parse the XML specification, we will look at the key diagram and the actual physical domain of the problem in Figure 4.16. Please note that zone 4 and 6 in the key diagram are void zones so no elements will be generated inside these two zones.

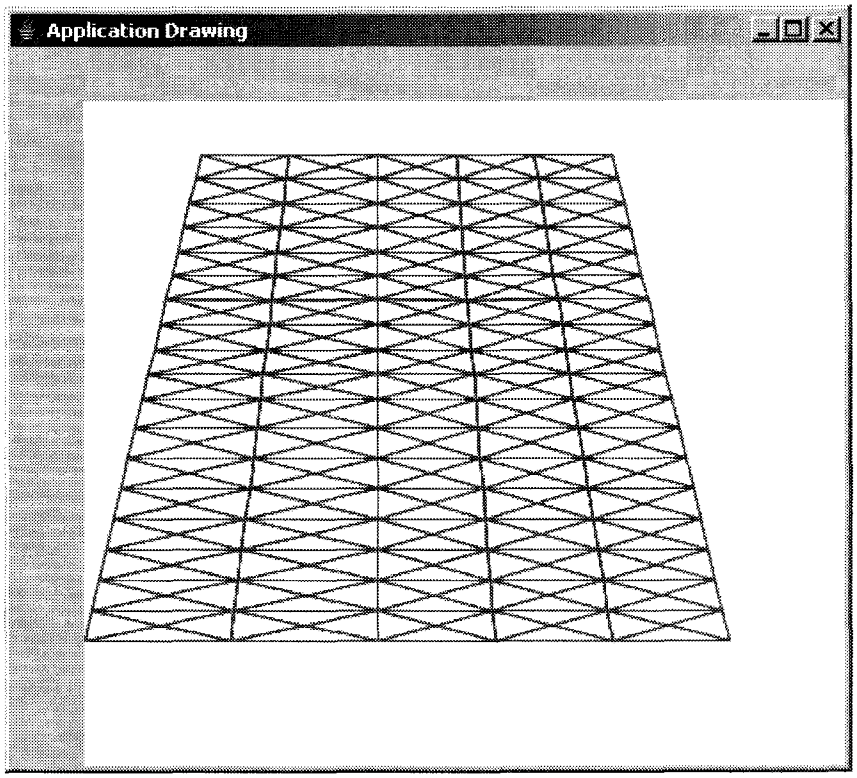


Figure 4.11: Generated Triangular Mesh by Changing the Element Shape and Topology

The XML specification in this case will provide the geometry and element information. The other parameters of variation will be set at run-time. The geometry and element information in the specification are provided in Figure 4.17 and 4.18, respectively.

4.2.1 Special-purpose vs. General Purpose

Parsing the specification of this example produces the same interface as in Figure 4.3. However, the differences between family members are reflected by the appearances of the dialogs for the specific category of information specified at specification time. For example, the geometry dialog in this mesh generator is shown in Figure 4.19, and the element information is shown in Figure 4.20.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
<xsl:output method="text"/>
<xsl:variable name="newline"><xsl:text>
</xsl:text></xsl:variable>
<xsl:variable name="tab"><xsl:text>&#x09;</xsl:text></xsl:variable>
<!-- category names -->
<xsl:template match="Mesh">
<xsl:text>ID&#x09;x_coord&#x09;y_coord&#x09;</xsl:text>
<xsl:value-of select="$newline"/>
<xsl:apply-templates select = "VertexSet"/>
</xsl:template>
<xsl:template match="VertexSet">
<xsl:for-each select = "Vertex">
<xsl:value-of select = "id"/> <xsl:value-of select = "$tab"/>
<xsl:value-of select = "x_coord"/> <xsl:value-of select = "$tab"/>
<xsl:value-of select = "y_coord"/> <xsl:value-of select = "$tab"/>
<xsl:value-of select = "$newline"/>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Figure 4.12: Stylesheet File 1

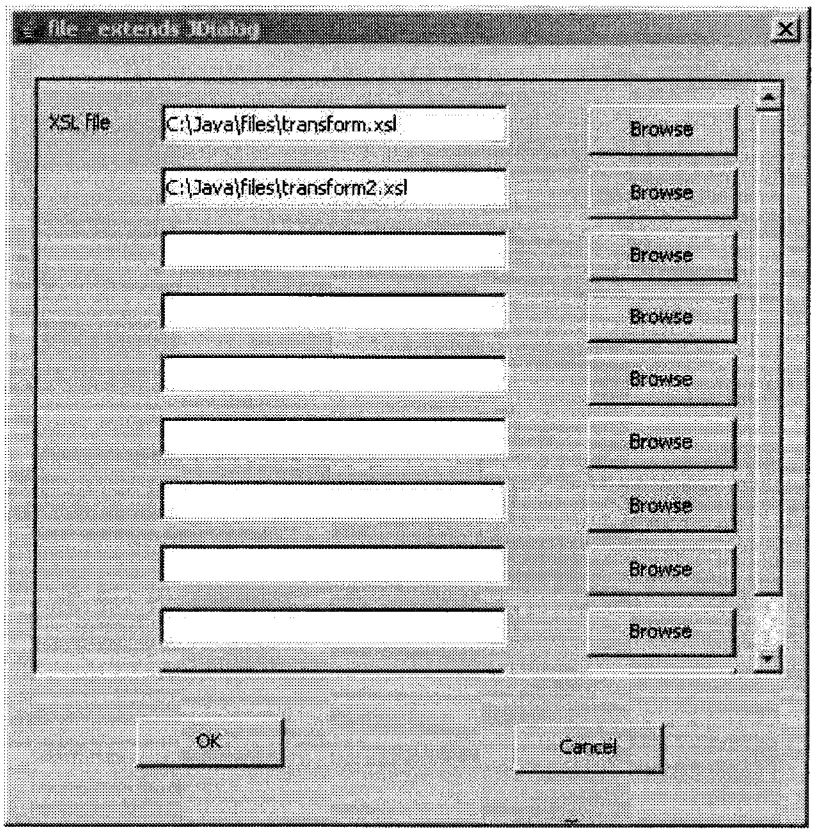
The values set at specification-time are passed into the interface and the values cannot be changed through the interface. This is how we can distinguish a general-purpose mesh generator from a special-purpose one. The general-purpose mesh generator allows every input to be specified at run time, while the special-purpose mesh generator fixes certain values at specification time. If two specifications differ in the values of some input, they will also produce different special-purpose family members. In this example, we can see that the geometry and element dialog have been fixed. The other input dialogs will be general-purpose and require user input information before a mesh can be generated.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
<xsl:output method="text"/>
<xsl:variable name="newline"><xsl:text>
</xsl:text></xsl:variable>
<xsl:variable name="tab"><xsl:text>&#x09;</xsl:text></xsl:variable>
<!-- category names -->
<xsl:template match="Mesh">
<xsl:text>ID&#x09;x_coord&#x09;y_coord&#x09;</xsl:text>
<xsl:value-of select="$newline"/>
<xsl:apply-templates select = "ElementSet"/>
</xsl:template>
<xsl:template match="ElementSet">
<xsl:for-each select = "Element">
<xsl:value-of select = "id"/> <xsl:value-of select = "$tab"/>
<xsl:value-of select = "element_connectivity"/>
<xsl:value-of select = "$newline"/>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Figure 4.13: Stylesheet File 2

4.2.2 Nonrectangular Domain Mesh Generation

After all the seed information is specified, the resulting mesh on the geometric domain is shown in Figure 4.21. Please note the element type has been bound at specification time to be “Triangle1”.



C:\Java\files\transform.xsl

C:\Java\files\transform2.xsl

Figure 4.14: Specified Information in Output File Dialog

```
ID  x_coord y_coord
1   0.0     0.0
2   0.5     0.0
3   1.0     0.0
4   1.4     0.0
5   1.8     0.0
6   2.2     0.0
7   0.025   0.125
8   0.5125  0.125
9   1.0     0.125
.       .           .
.       .           .
.       .           .
```

Figure 4.15: Output File 1

```
ID   connectivity
ID   connectivity
1      0 1 7 6
2      1 2 8 7
3      2 3 9 8
4      3 4 10 9
5      4 5 11 10
6      6 7 13 12
7      7 8 14 13
8      8 9 15 14
9      9 10 16 15
.      .  .  .  .
.      .  .  .  .
.      .  .  .  .
```

Figure 4.16: Output File 2

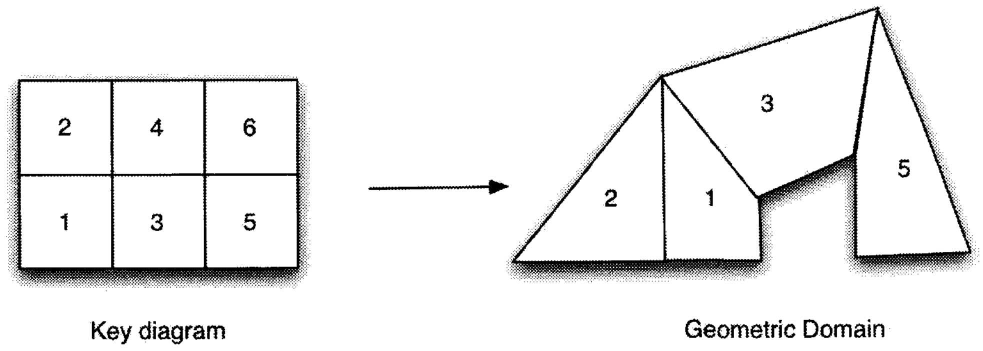


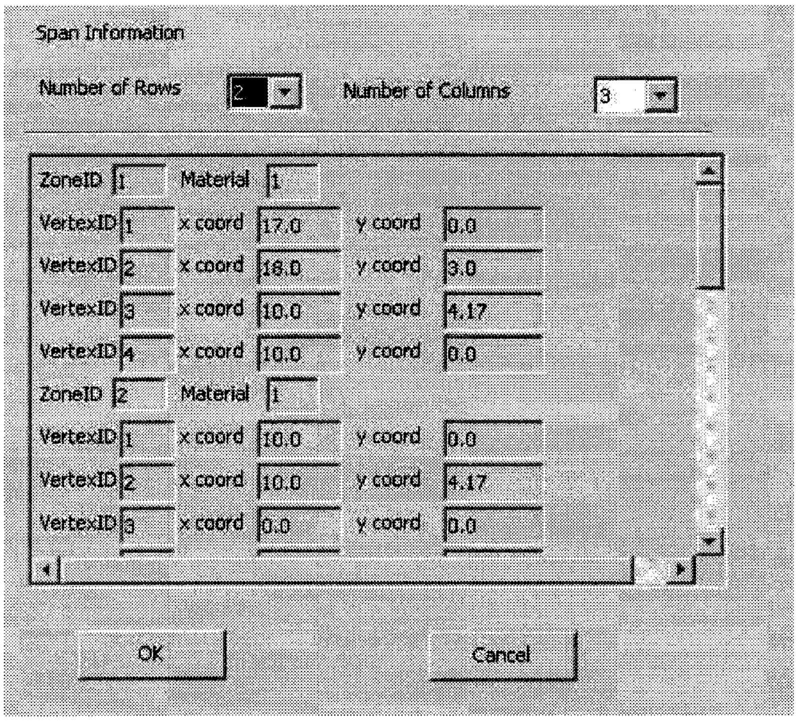
Figure 4.17: Key Diagram and Geometric Domain

```
<geometry>
        <spanSpec>
            <rows>2</rows>
            <cols>3</cols>
        </spanSpec>
        <zoneSpec ID="1" MaterialAssigned="1">
            <vertex ID="1">
                <x_coord>17</x_coord>
                <y_coord>0</y_coord>
            </vertex>
            <vertex ID="2">
                <x_coord>18</x_coord>
                <y_coord>3</y_coord>
            </vertex>
            <vertex ID="3">
                <x_coord>10</x_coord>
                <y_coord>4.17</y_coord>
            </vertex>
            <vertex ID="4">
                <x_coord>10</x_coord>
                <y_coord>0</y_coord>
            </vertex>
     </zoneSpec>
     .
     .
     . continued for six zones
</geometry>
```

Figure 4.18: XML specification of Geometry

```
<elementSet>
        <geometrySpec>
            <shape>triangle1</shape>
            <subdivisions>
          <raxis>2,2</raxis>
                <saxis>2,3,2</saxis>
            </subdivisions>
            <pattern>triangle1</pattern>
            <nodeGeo count="3">
                <node id="1">
                    <location>1,0,0</location>
                </node>
                <node id="2">
                    <location>0,1,0</location>
                </node>
                <node id="3">
                    <location>0,0,1</location>
                </node>
            </nodeGeo>
        </geometrySpec>
</elementSet>
```

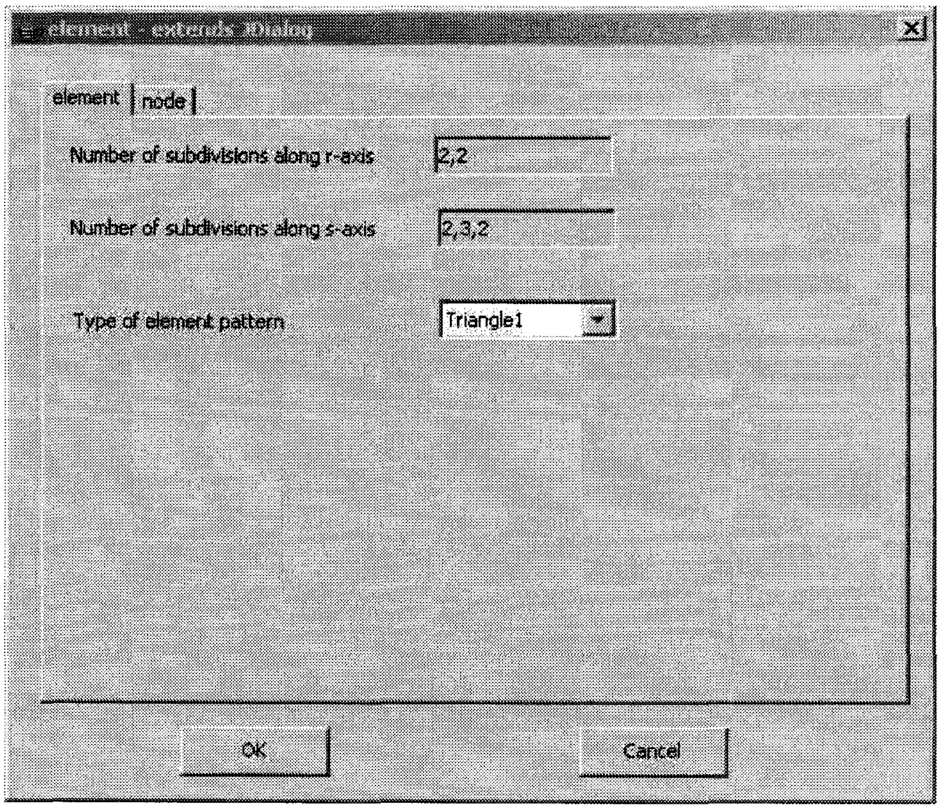
Figure 4.19: XML specification of Element Type


Span Information

Number of Rows 2 Number of Columns 3

ZoneID	1	Material	1		
VertexID	1	x coord	17.0	y coord	0.0
VertexID	2	x coord	18.0	y coord	3.0
VertexID	3	x coord	10.0	y coord	4.17
VertexID	4	x coord	10.0	y coord	0.0
ZoneID	2	Material	1		
VertexID	1	x coord	10.0	y coord	0.0
VertexID	2	x coord	10.0	y coord	4.17
VertexID	3	x coord	0.0	y coord	0.0

Figure 4.20: Geometry Dialog Appearance for Specification-time Information



Number of subdivisions along r-axis: 2,2

Number of subdivisions along s-axis: 2,3,2

Type of element pattern: Triangle1

Figure 4.21: Geometry Dialog Appearance for Specification-time Information

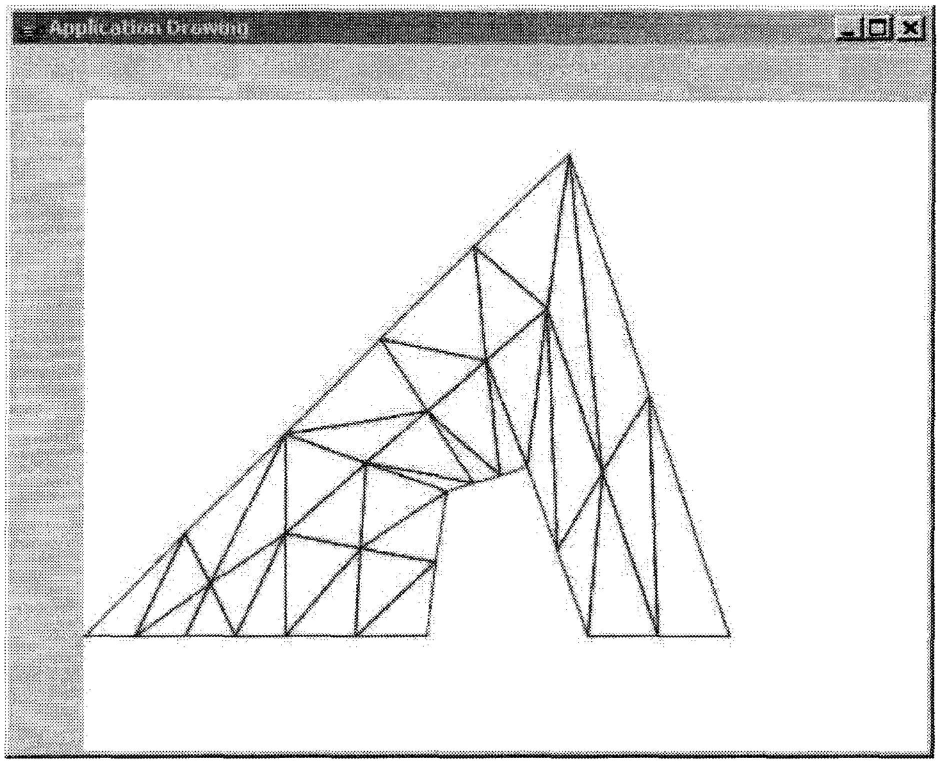


Figure 4.22: Triangular Mesh Produced by Special-purpose Mesh Generator

Chapter 5

Conclusions

In this chapter we first present the conclusions of our work in Section 5.1. We then list the contributions of our work in Section 5.2 and offer suggestions for future work in Section 5.3.

5.1 Conclusions

In this thesis we presented PMG as a framework for rapid development of special-purpose mesh generators as members of a program family. The system design process involves producing the following documents: a commonality analysis, a requirement specification, and a module guide. We also implemented a prototype of PMG. From this experience, the following conclusions can be made:

[1] Special-purpose mesh generators are suitable for development as a program family. The most important result we have shown in our work is that the program family design is a good approach for rapid development of special-purpose mesh generators. Although the nature of mesh generators as software suggests that they are ideal candidates to be developed as a program family [47], the implementation of PMG

clearly displays our confidence that not only is the appropriateness sound in theory, it can also be demonstrated through a proof of concept implementation. As shown in Chapter 4, the variabilities between different family members are reflected by fixing the values of parameters of variation at specification time or run time.

[2] Commonality analysis greatly assisted in developing PMG and it should be equally beneficial for developing other family-based scientific computing software. The key in family-oriented software development is the prediction of future changes. In the design of PMG, this is handled by conducting a commonality analysis. A commonality analysis is an analytical approach that consists of systematically identifying and documenting the commonalities that all program family members share, the variabilities between family members and the terminology used in describing the family. It was initially proposed by Weiss [40] and has been demonstrated in FAST (Family oriented Abstraction, Specification and Translation) [13, 52]. Our design process spent considerable time refining an original commonality analysis documented by Smith and Chen [47] to reflect our scope time decisions. Our experience in the end is that a good commonality analysis is not only a successful starting point for the subsequent design activities, such as requirement specifications and module guide, but also provides the guidance for the implementation of the system to model different mesh generator members in the family. For example, the DSL design follows the commonality analysis such that the values of each variability bound at specification time can be specified in the DSL.

[3] A module guide with clear module decomposition is beneficial to the system development. The module guide documented in Appendix C defines the structures of the system and was used to guide our implementation consistently through the programming stage. The module decomposition shown in the module guide are based on

the information hiding principle and separation of concern. The anticipated changes from the commonality analysis are encapsulated in separate modules; therefore, each module hides some design decisions of the system. The module guide also provides the relationship of modules by displaying the "uses" hierarchy, so that the responsibilities and functions of the modules are clear.

[4] The concept of "Design through documentation" guides the entire process. Documentation of software design serves as a media for communication between developers, and can also be used to verify and test the system functions based on the functional requirements in the requirement specifications. All the design documents included in the appendices are a good source to understand PMG and for possibly maintaining and further development in the future. These documents and the traceability that exists between them greatly improve the internal quality of maintainability.

5.2 Contributions of Our Work

This section lists the achievements of our work as follows:

[1] Presenting a framework for rapid development of special-purpose mesh generators.

[2] Revising an existing commonality analysis to reflect the scope time decision.

[3] Documenting a software requirement specification for the program family.

[4] Applying module decomposition techniques and documenting a module guide.

[5] Using XML to implement a domain-specific language to model the seed information necessary to produce a mesh generator.

[6] Applying advanced Java parsing techniques to customize the object representing a mesh generator family member.

[7] Programming Java Swing components for the graphical interface of the mesh generators.

[8] Implementing a meshing algorithm introduced by Zienkiewicz and Phillips [56] to produce structured meshes with quadrilateral and triangular elements.

[9] Using XSL stylesheets to allow flexible customization of the output file(s).

[10] Providing a mechanism to quantify the degree of generality of a mesh generator, since the DSL for a program family member explicitly shows these parameters of variation that are fixed at specification time versus those that are set at run time.

5.3 Future Work

The results of our work encourage future research in the development of a family of mesh generators. Our time constraints have limited us to only implement a simple version of PMG. Therefore, more work should be done in the future to refine the program and expand the scope of the thesis. A list of possible further investigations is as follows:

– Expand the program family. Many refinements can be done by revising the scope time decisions in this thesis to specification time or run time binding. This include programming work with respect to satisfying functional requirements and nonfunctional requirements.

  [1] In terms of functional requirements, many scope time decisions in the commonality analysis can be reused to cover more possibilities. For example, the future

versions of PMG can include the support of unstructured mesh generation and mesh optimization features (e.g. mesh refinement and smoothing).

[2] Improve the user interface for better usability. For example, future versions of PMG can add mesh editing features and allow the geometry to be specified by drawing the bounding domain with a mouse.

[3] Improve the robustness of PMG. Any error in the user inputs including inconsistencies between inputs, should be detected and handled before generating the mesh. For example, the GUI should check that a coordinate of the geometry only contains numbers. A more complex example may be that the number of degrees of freedom should be the same as the number of values of fixity.

– A module interface specification (MIS) should be conducted using formal mathematics and logic. An MIS documents the design secrets and key functions of each module in the system. In other words, an MIS describes "what a module does, but not how it does it." The module specification in the MIS also explains how the current modules interact with other modules by exporting some functions or variables to be used.

– The program family approach demonstrated in this thesis may be used to create other scientific computing software. For instance, a family of finite element analysis programs or ordinary differential equation (ODE) solvers may be developed using a similar approach as that shown in this thesis.

Bibliography

[1] I. S. 1233. Ieee guide for developing system requirements specifications, 1996.

[2] M. Ardis and D. M. Weiss. Defining families: The commonality analysis. In *Nineteenth International Conference on Software Engineering*. ACM, Inc., 1997.

[3] R. E. Bank. A domain decomposition for a parallel adaptive meshing algorithm. Department of Mathematics University of California, San Diego La Jolla, California, 92093–0112, 2002.

[4] M. Bern and P. Plassmann. *Mesh Generation. Handbook of Computational Geometry.* Elsevier Science, 2000.

[5] G. Berti. Generic components for grid data structures and algorithms with C++. *In First Workshop on C++ Template Programming*, October 2000.

[6] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[7] M. Cecilia and N. Hitschfeld-Kahler. An evolvable meshing tool through a flexible

object-oriented design. *Proceedings, 13th International Meshing Roundtable*, SAND 2004–3765C:pp.203–212, September 2004.

[8] C.-H. Chen. A software engineering approach to developing mesh generators. *McMaster University, Hamilton, Ontario, Canada*, 2003.

[9] L. Chen. Mesh smoothing schemes based on optimal delaunay triangulations. *Proceedings of 13th International Meshing Roundtable*, pages 109–120, September 19–22, 2004.

[10] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In *Proceedings of the 10th International Symposium on Programming Languages, Implementations, Logics and Programs PLILP/ALP '98*, 1998. Invited paper.

[11] W. Consortium. Mathml 2.0. *URL: http://www.w3c.org/Math.*

[12] W. consortium. The extensible stylesheet language family (XSL). *URL:www.w3.org/Style/XSL*, 2005.

[13] D. A. Cuka and D. M. Weiss. Specifying executable commands: An example of fast domain engineering. *Submitted to IEEE Transactions on Software Engineering*, pages 1–12, 1997.

[14] R. de Groot. Design and implementation of embedded domain-specific languages. *Center for Software Technology, Institute of Information and Computing Sciences, Utrecht University*, 2005.

[15] E. W. Dijkstra. *Structured Programming, chapter Notes on Structured Programming.* Academic Press,London, 1972.

[16] M. J. Dino Mandrioli, Carlo Ghezzi. *Fundamentals of Software Engineering.* Prentice–Hall, USA, 2005.

[17] A. H. ElSheikh, W. S. Smith, and S. E. Chidiac. Semi-formal design of reliable mesh generation systems. *Advances in Engineering Software*, 35(12):827–841, 2004.

[18] P. J. Frey and P. L. George. *Mesh Generation Application to Finite Elements.* Hermes Science Europe Ltd., 2000.

[19] R. V. Garimella. Mstk - a flexible infrastructure library for developing mesh based applications. *Proceedings, 13th International Meshing Roundtable*, SAND 2004–3765C:pp.203–212, September 2004.

[20] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *http://www.geuz.org/gmsh*, November 2005. Version 1.61.

[21] D. H. R. Gopalsamy, S. and A. M. Shih. Api for grid generation over topological models. *Proceedings, 13th International Meshing Roundtable*, SAND 2004–3765C:pp.221–230, September 2004.

[22] X. group. Matml, xml for material property data. *URL: http://www.matml.org*, 2005.

[23] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28A, December 1996.

[24] M. S. J.F. Remacle, B.K. Karamete. *Algorithm oriented mesh database.* 2000.

[25] P. H. John Peterson and C. Elliott. Lambda in motion: Controlling robots with haskell. *First International Workshop on Practical Aspects of Declarative Languages*, 1999.

[26] S. C. Johnson. Yacc: Yet another compiler-compiler. *Computing Science Technical Report 32*, 1975. AT&T Bell Laboratories, Murray Hill NJ.

[27] S. C. Johnson. Lint, a c program checker. *Comp. Sci. Tech. Rep.*, (No. 65), 1978.

[28] S. C. Johnson. A portable compiler: Theory and practice. *Proc. 5th ACM Symp. on Principles of Programming Languages*, pages pp. 97–104, January 1978.

[29] R. E. Jones. The qmesh mesh generation package. *Proceedings of the SIGNUM meeting on Software for partial differential equations*, pages 31–34, 1975.

[30] S. Kamin and D. Hyatt. A special-purpose language for picture-drawing. *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1999.

[31] B. W. Kernighan and L. L. Cherry. A system for typesetting mathematics. *Comm. Assoc. Comp. Mach.*, pages pp. 151–157, March 1975. Bell Laboratories, Murray Hill, New Jersey.

[32] D. Leijen. wxhaskell – a portable and concise gui library for haskell. *In ACM SIGPLAN Haskell Workshop (HW'04). ACM Press*, September 2004.

[33] M. E. Lesk and E. Schmidt. *lex: A lexical analyzer generator*, volume 2, pages 388–400. Holt, Rinehart, and Winston, New York, NY, USA, 1979. In UNIX Programmer's Manual.

[34] B. McLaughlin. *Java & XML data binding.* O'reilly, 2002.

[35] Sun MicroSystems. Java architecture for xml binding (jaxb) . *URL: http://java.sun.com/webservices/jaxb*, 2005.

[36] L. F. M.J. and P. P. An efficient parallel algorithm for mesh smoothing. *4th International Meshing Roundtable*, pages pp. 47–58, 1995.

[37] P. Murray-Rust and H. S. Rzepa. Chemical markup language. *http://www.xml-cml.org/information/position.html*, 2001. A Position Paper.

[38] C. Ollivier-Gooch. Grummp. *URL: http://tetra.mech.ubc.ca/GRUMMP /index.html*, 2002.

[39] S. Owen. Meshing research corner. *http://www.andrew.cmu.edu/user/sowen/mesh.html*, May 2006.

[40] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, March 1976.

[41] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, pages 128–138, March 1979.

[42] D. L. Parnas. Why software jewels are rare. *IEEE Computer*, 29:57–60, 1996.

[43] A. Project. Xalan, the xslt processor. *URL: www.xml.apache.org/xalan-j/*, 2005.

[44] M. Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 25(3):291–315, 2003.

[45] R. Schneider. Meshing software. *URL: http://www-users.informatik.rwth-aachen.de/ roberts/software.html*, May 2006.

[46] SCOREC. Aomd documentation. *URL: www.scorec.rpi.edu/AOMD/*

[47] W. S. Smith and C.-H. Chen. Commonality and requirements analysis for mesh generating software. In *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, pages 384–387, Banff, Alberta, Canada, June 2004.

[48] S.Smith and C.H.Chen. Commonality analysis for mesh generating systems. Technical report, Computing and Software Department,McMaster University, 2004.

[49] D. Stolle. Grid generator. *Department of Civil Engineering, McMaster University*, 2000.

[50] R. H. Thayer and M. Dorfman. Ieee recommended practice for software requirements specifications. *IEEE Computer Society*, 2000.

[51] N. R. o. C. The Numerical Modelling Group Mining Research Laboratories. Cansafe / visrock - a windows application packlage for stress analysis using finite element techniques. *Division Report: MRL 94-046(TR)*, October 1994.

[52] D. Weiss and C. Lai. *Software Product Line Engineering*. Addison–Wesley, 1999.

[53] D. M. Weiss. Defining families: The commonality analysis. *Submitted to IEEE Transactions on Software Engineering*, 1997.

[54] D. M. Weiss. Commonality analysis: A systematic process for defining families. *Lecture Notes in Computer Science*, pages 214–222, 1998.

[55] wxWidget Project. Wxwidget cross-platform gui library. *URL:http://www.wxwidgets.org/*, 2005.

[56] O. Zienkiewicz and D. V. Phillips. An automatic mesh generation scheme for plane and curved surfaces by 'isoparametric' co-ordinates. *International Journal for Numerical Methods in Engineering*, pages 519–528, 1971.

Appendix A

Commonality Analysis for Mesh Generating Systems

A.1 Introduction

This document serves as the starting point to designing mesh generators as a program family. The targeted family is the type of mesh generators that are pre-processors to finite element programs. An existing commonality analysis for the same type of mesh generators has been conducted and documented by Smith and Chen [48]. However, our research scope is more restricted than Smith and Chen [48] because we only focus on structured mesh generators on 2D domains. Therefore, we use the results from Smith and Chen [48] as the basis of our document, while making necessary changes to reflect the change of scope. For the convenience of referring to the original commonality analysis results, we use the abbreviation S&C to designate the Smith and Chen document.

The current document is organized as follows: Section 2 contains the list of terminology definitions and abbreviations used in the document. Section 3 records the commonalities shared by the members of the mesh generator family. Section 4 and 5 record the variabilities and the corresponding parameters of variation, respectively.

A.2 Terminology and Definitions

This section lists the terminologies used in the field of software engineering and mesh generation. Most terminologies associated with our commonality analysis has been provided in S&C, to which the reader can refer for the definitions. The definition given below is the only one that differs from S&C. The change was made to better integrate the notions of connectivity of a mesh element versus connectivity of a mesh.

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

[1] "The connectivity of a mesh element is the definition of the connection between the vertices at the element level." [18]

[2] The connectivity of the mesh is given by the set of connectivities of its constituent elements.

A.3 Commonalities

This section lists all the common features among all the potential family members. The commonalities are organized using the same abstraction of the system as indicated in [48]. The categories of the abstraction are as follows: input information, mesh generation, and output information. Section A.3.1 describes the commonalities for the mesh generation step, which includes the discretization of the domain, as well as other information on the problem such as the boundary conditions, material properties, etc. Section A.3.2 highlights the input information that is required for all mesh generators, such as the geometry of the domain that is going to be discretized. The next section, Section A.3.3, shows the common features for the output of mesh generators, such as the requirement that mesh information be written to files. The final section covers qualities of the system that cannot be classified as input, mesh generation or output. These commonalities are termed nonfunctional requirements of the system.

Each commonality below uses the same structure. All of the commonalities are assigned a unique item number, which takes the form of a natural number with the prefix "C". Following this, a description of the commonality is provided along with a list of related variabilities, which are given as hyperlinks that allow navigation of the document to the text describing the variability. Finally, each commonality ends with a summary of the history, including the date of creation and any dates of modification, along with a brief description of the modification. If a commonality is borrowed from another document such as S&C, then the corresponding history section will mention its source.

Since we restricted our scope to cover a less broad range of mesh generators, some variabilities need to be redefined and become commonalities in the current commonality analysis. These new commonalities are added in this section, where for each new commonality, its original source of variability will be mentioned in the corresponding history field.

A.3.1 Mesh Generation

Item Number	C1
Description	A mesh generator discretizes a given computational domain (closed boundary Ω) into a covering up of a finite number of simple shapes.
Related Variability	V4, V8, V10
History	Borrowed from S&C – May 9, 2005

Item Number	C2
Description	Each vertex has a unique identifier.
Related Variability	None
History	Borrowed from S&C – May 9, 2005

Item Number	C3
Description	Each element has a unique identifier.
Related Variability	None
History	Borrowed from S&C – May 9, 2005

Item Number	C4
Description	An element's topology is given by the connectivity of its set of vertices.
Related Variability	None
History	Borrowed from S&C – May 9, 2005

Item Number	C5
Description	Information on the created meshes includes material properties.
Related Variability	V11, V12
History	Borrowed from S&C – May 9, 2005

Item Number	C6
Description	Information on the created meshes includes boundary conditions.
Related Variability	V13
History	Borrowed from S&C – May 9, 2005

Item Number	C7
Description	Information on the created meshes includes system parameters, such as the number of elements in the domain and numerical parameters needed by the finite element analysis program.
Related Variability	V14
History	Borrowed from S&C – May 9, 2005

Item Number	C8
Description	The mesh generators will not provide optimization features such as smoothing, and refinement/coarsening.
Related Variability	None
History	Created – May 9, 2005. Modified from variability V3 from S&C

Item Number	C9
Description	The mesh generators will not provide mesh editing features, such as “tweaking” a vertex location, *etc.*
Related Variability	None
History	Created – May 9, 2005. Modified from variability V4 from S&C

Item Number	C10
Description	The vertices in the mesh will be numbered in the order that minimizes bandwidth.
Related Variability	None
History	Created - May 9, 2005. Modified from variability V5 from S&C

Item Number	C11
Description	From topological perspective, all mesh generators in our scope produce only structured meshes.
Related Variability	V4, V8, V10
History	Created - May 9, 2005. This commonality is modified from V6 from S&C

Item Number	C12
Description	The local numbering of vertices and nodes in the mesh is always counter-clockwise.
Related Variability	None
History	Created - May 9, 2005. Modified from variability V8 from S&C

Item Number	C13
Description	The mesh generators will not accommodate a mixed mesh, a hybrid mesh, or a nonconformal mesh.
Related Variability	None
History	Created - May 9, 2005. Modified from variabilities V16–V18 from S&C

Item Number	C14
Description	The mesh generators only use Cartesian coordinates to describe the geometry.
Related Variability	V10
History	Created - May 9, 2005. Modified from variabilities V20 from S&C

A.3.2 Input

Item Number	C15
Description	A mesh generator requires that information be input by the user to define his/her meshing problem.
Related Variability	V14
History	Borrowed from S&C – May 9, 2005

Item Number	C16
Description	The mesh generators provide a graphical user interface.
Related Variability	None
History	Created - May 9, 2005. Modified from variabilities V21 from S&C

Item Number	C17
Description	The interface for specifying the closed boundary of the domain is the selection of the number of subdivisions along each direction.
Related Variability	None
History	Created - May 9, 2005. Modified from variability V22 from S&C

Item Number	C18
Description	The mesh generators will not allow the specification of two degrees of freedoms to have the same value.
Related Variability	None
History	Created - May 9, 2005. Modified from variability V28 from S&C

Item Number	C19
Description	The mesh generators will not allow the specification of internal boundaries.
Related Variability	None
History	Created - May 9, 2005. Modified from variability V29 from S&C

Item Number	C20
Description	The user defines the geometric domain of the problem using the same approach as Zienkiewicz and Phillips.
Related Variability	None
History	Created - May 9, 2005. This commonality is modified from V23 from S&C

Item Number	C21
Description	The user needs to specify the physical attributes, such as the material properties, the boundary conditions, *etc.*
Related Variability	V11, V12, V13, V14
History	Borrowed from S&C – May 9, 2005

Item Number	C22
Description	When boundary conditions are specified, a maximum of one condition may be given for each degree of freedom (dof). For instance, a dof cannot have both a prescribed displacement and a prescribed force.
Related Variability	V13
History	Borrowed from S&C – May 9, 2005

A.3.3 Output

Item Number	C23
Description	The mesh generators display the resulting mesh on the screen.
Related Variability	None
History	Created – May 9, 2005. Modified from variability V30 from S&C

Item Number	C24
Description	Mesh generators write mesh information to text file(s).
Related Variability	V15, V16
History	Borrowed from S&C - May 9, 2005

Item Number	C25
Description	The element information of a mesh is listed in the output file in some order.
Related Variability	V17
History	Borrowed from S&C – May 9, 2005

Item Number	C26
Description	The vertex information, such as the coordinates, for a mesh is listed in output file(s) in some order.
Related Variability	V18
History	Borrowed from S&C – May 9, 2005

A.3.4 Nonfunctional Requirements

Item Number	C27
Description	The mesh generators should perform the essential functions in reasonable time, such as essentially instantaneously for entering user input, seconds for waiting graphics display, or minutes for mesh generation and output file creation.
Related Variability	None
History	Created - May 9, 2005; Modified from variability V39 from S&C

Item Number	C28
Description	The mesh generator will provide reasonable measures of accuracy consistent with usual expectations for engineering and scientific software.
Related Variability	None
History	Created - May 9, 2005; Modified from variability V40 from S&C

Item Number	C29
Description	The mesh generator provides the precision consistent with usual expectations for engineering and scientific software.
Related Variability	None
History	Created - May 9, 2005; Modified from variability V41 from S&C

Item Number	C30
Description	The quality of generated mesh, measured in attributes such as aspect ratio, minimum angle, *etc.* is consistent with those qualities that are generally true of engineering computing software.
Related Variability	None
History	Created - May 9, 2005; Modified from variability V15 from S&C
Item Number	C31
Description	The mesh generator is robust enough to handle the types of users and the types of problems that the system is expected to encounter.
Related Variability	None
History	Borrowed from S&C – May 9, 2005

Item Number	C32
Description	The mesh generator will be as portable to other operating systems as required by the users of the system.
Related Variability	V20
History	Borrowed from S&C - May 9, 2005

Item Number	C33
Description	The mesh generators require reasonable amount of memory and disk storage, consistent with the resources of a typical personal computer in the year 2005.
Related Variability	None
History	Created - May 9, 2005; Modified from variability V38 from S&C

A.4 Variabilities

This section provides a list of characteristics that may vary among family members. As in Section A.3, the first three subsections on variabilities are organized into the following sublists: Mesh Generation, Input and Output. The final two subsections list variabilities that can be characterized as system constraints and as nonfunctional requirements.

As for the commonalities, each variability is labeled with a unique item number. In this case the numbers are preceded with the letter "V". The other four headings provided for each variability are: Description, Related Commonality, Related Parameter and History. The related commonalities and parameters are given as a set of identifiers that respectively refer back to the previous section on commonalities or refer forward to the next section on parameters of variation.

A.4.1 Mesh Generation

Item Number	V1
Description	Different mesh generators will be able to accommodate the creation of meshes for different problem domains.
Related Commonality	None
Related Parameter	P1
History	Borrowed from S&C – May 9, 2005

Item Number	V2
Description	The degree of generality of the mesh generator. Some mesh generators are general purpose programs, while others are tailored to a specific application domain.
Related Commonality	None
Related Parameter	P2
History	Borrowed from S&C – May 9, 2005

Item Number	V3
Description	For structured meshes, different templates for the local patterns in the element topology are possible.
Related Commonality	C1, C11
Related Parameter	P3
History	Borrowed from S&C – May 9, 2005

Item Number	V4
Description	The shape of the elements generated by the mesh generator as defined by their vertices.
Related Commonality	C1, C11
Related Parameter	P4
History	Borrowed from S&C – May 9, 2005

Item Number	V5
Description	The number of nodes for an element and the location of those nodes.
Related Commonality	None
Related Parameter	P5
History	Borrowed from S&C – May 9, 2005

Item Number	V6
Description	The number of degrees of freedom at a node and the meaning of each of those degrees of freedom.
Related Commonality	None
Related Parameter	P6
History	Borrowed from S&C – May 9, 2005

Item Number	V7
Description	The pattern of the number of degrees of freedom and the meaning of these degrees of freedom can vary between the nodes of an element.
Related Commonality	None
Related Parameter	P7
History	Borrowed from S&C – May 9, 2005

Item Number	V8
Description	The dimensionality of the computational domain.
Related Commonality	C1
Related Parameter	P8
History	Borrowed from S&C – May 9, 2005

Item Number	V9
Description	The shape allowed for the computational domain.
Related Commonality	C1, C20
Related Parameter	P9
History	Borrowed from S&C – May 9, 2005

Item Number	V10
Description	The dimension of the coordinate system used to describe the geometry (coordinates) of the vertices and possibly of the nodes.
Related Commonality	C1, C20, C14
Related Parameter	P10
History	Borrowed from S&C – May 9, 2005

A.4.2 Input

Item Number	V11
Description	The number of material properties, their names and their types.
Related Commonality	C5, C21
Related Parameter	P11
History	Borrowed from S&C – May 9, 2005

Item Number	V12
Description	The number of different materials allowed in the specification of the physical problem.
Related Commonality	C5, C21
Related Parameter	P12
History	Borrowed from S&C – May 9, 2005

Item Number	V13
Description	The types of boundary conditions accommodated by the system.
Related Commonality	C6, C21, C22
Related Parameter	P13
History	Borrowed from S&C – May 9, 2005

Item Number	V14
Description	The number and type of different system parameters input to the system. These parameters will be passed on to the finite element program.
Related Commonality	C7, C15, C21
Related Parameter	P14
History	Borrowed from S&C – May 9, 2005

A.4.3 Output

Item Number	V15
Description	The number of files that are output by the mesh generator.
Related Commonality	C24
Related Parameter	P15
History	Borrowed from S&C – May 9, 2005

Item Number	V16
Description	The format of the information in the file(s) output by the mesh generator.
Related Commonality	C24
Related Parameter	P16
History	Borrowed from S&C – May 9, 2005

Item Number	V17
Description	The element information is written to the file(s) following a different ordering.
Related Commonality	C25
Related Parameter	P17
History	Borrowed from S&C – May 9, 2005

Item Number	V18
Description	The vertex information is written to the file(s) following a different ordering.
Related Commonality	C26
Related Parameter	P18
History	Borrowed from S&C – May 9, 2005

Item Number	V19
Description	The degree to which the user can customize the output file formats.
Related Commonality	C24
Related Parameter	P19
History	Borrowed from S&C – May 9, 2005

A.4.4 System Constraints

Item Number	V20
Description	The operating systems on which the mesh generating system is intended to run.
Related Commonality	C32
Related Parameter	P20
History	Borrowed from S&C – May 9, 2005

A.5 Parameters of Variation

This section specifies the parameters of variation for the variabilities listed in Section A.4. They are organized into the same five subcategories as employed previously: Mesh Generation, Input, Output, System Constraints, Nonfunctional Requirements.

Each parameter of variation is given a unique identifier of the form “P” followed by a natural number. The corresponding variability is listed to allow navigation back to the appropriate item in Section A.4. The final entry for each parameter of variation is the binding time, which is the time in the software lifecycle when the variability is fixed. The binding time could be during specification, or during building of the system (compile time), or during execution of the system (run time). It is possible to have a mixture of binding times. For instance, a parameter of variation could have a binding time of "specification or building" to represent that the parameter could be set at specification time, or it could be postponed until the given family member is built. The choice of postponing the decision until the build would be associated with the presence of a domain specific language that would allow postponing decisions on the values of the parameter of variation.

A.5.1 Mesh Generation

Item Number	P1
Corresponding Variability	V1
Range of Parameters	Mesh generating systems can build meshes for a large range of problem domains corresponding to the large range of problems that can be solved via finite element analysis. For instance, the mesh data files can be targeted toward the following problem domains: solid mechanics, fluid mechanics, heat transfer, seepage, electrostatics, *etc.*
Binding Time	specification or run time

Item Number	P2
Corresponding Variability	V2
Range of Parameters	A continuum exists from the most specialized systems to arbitrarily general systems. For instance, a special purpose system may involve quadrilateral elements on a 2D rectangular domain for the purpose of solving for the temperature in a heated plate. On the other hand, a general purpose system would handle an arbitrary geometry for the domain, provide a choice of element shapes, allow for 1D, 2D or 3D domains and provide meshes for a variety of physical problems.
Binding Time	specification or build time

Item Number	P3
Corresponding Variability	V3
Range of Parameters	Nine(9) potential local topology templates are possible, as shown in Appendix A.
Binding Time	specification or build or run time

Item Number	P4
Corresponding Variability	V4
Range of Parameters	In 1D there are line segments; in 2D there are triangles and quadrilaterals.
Binding Time	specification or build or run time

Item Number	P5
Corresponding Variability	V5
Range of Parameters	The element can have fewer nodes than vertices, the same number of nodes as vertices or more nodes than vertices. The nodes can be located at the vertices, on the element edges, or inside the element.
Binding Time	specification or build or run time

Item Number	P6
Corresponding Variability	V6
Range of Parameters	The number and type of degrees of freedom at the nodes can vary between different types of elements and within an element. The dof for an element represent the dependent variable that will be solved for. Some example dof are as follows: displacements, velocities, temperatures, voltages, pressures, *etc.*
Binding Time	specification or build or run time

Item Number	P7
Corresponding Variability	V7
Range of Parameters	If the geometry is interpolated at fewer nodes than the interpolation of the dof, then the element is subparametric. If the geometry is interpolated at the same number of nodes as the interpolation for the dof, then the element is isoparametric. If the geometry is interpolated at more nodes than the interpolation for the dof, then the element is superparametric.
Binding Time	specification or build or run time

Item Number	P8
Corresponding Variability	V8
Range of Parameters	1D or 2D
Binding Time	specification or build or run time

Item Number	P9
Corresponding Variability	V9
Range of Parameters	A 2D domain is of any possible shape allowed by Zienkiewicz and Phillips.
Binding Time	specification or build time

Item Number	P10
Corresponding Variability	V10
Range of Parameters	The dimension of the spatial coordinate system that is used to express the geometric coordinates. The only possible option is 2D.
Binding Time	specification or build or run time

A.5.2 Input

Item Number	P11
Corresponding Variability	V11
Range of Parameters	The number of material properties is variable and can include such properties as elastic modulus, viscosity, relaxation time, thermal conductivity, *etc.*
Binding Time	specification or build or run time

Item Number	P12
Corresponding Variability	V12
Range of Parameters	The entire domain can consist of one material or there may be any finite number of different materials.
Binding Time	specification or build or run time

Item Number	P13
Corresponding Variability	V13
Range of Parameters	The boundary conditions may be of the Dirichlet or Neumann. If the boundary conditions are for prescribed values (Dirichlet type) that are zero, they may be specified in a different manner from other prescribed values. For instance, in solid mechanics problems a boundary may be fixed in one or more directions so that it cannot move in that direction and it will be free in the remaining directions. The input may specify this kind of fixity information.
Binding Time	specification or build time

Item Number	P14
Corresponding Variability	V14
Range of Parameters	The number and meaning of the system parameters can vary from one mesh generator to the next. System parameters may include global numerical parameters, such as the degree of implicitness for a time marching scheme.
Binding Time	specification or build or run time

A.5.3 Output

Item Number	P15
Corresponding Variability	V15
Range of Parameters	The number of files can range from 1 to many. In the case of many files the data can be split between files, possibly so that geometry data, topology data, material properties data, *etc.* are separated.
Binding Time	specification or build or run time

Item Number	P16
Corresponding Variability	V16
Range of Parameters	Different mesh generators organize mesh information into file(s) in different orders. The data structure that is output can change between mesh generators, or it may be something that the user can customize within a given mesh generator.
Binding Time	specification, build or run time

Item Number	P17
Corresponding Variability	V17
Range of Parameters	Some mesh generators list elements in an increasing order (implicity), while other explicitly output the element identifier and list them in an arbitrary order.
Binding Time	specification or build or run time

Item Number	P18
Corresponding Variability	V18
Range of Parameters	Some mesh generators list nodal information in ascending order (implicitly), but others explicitly output the node's identifier and list them in an arbitrary order.
Binding Time	specification or build or run time

Item Number	P19
Corresponding Variability	V19
Range of Parameters	Some mesh generators will have a fixed file format, while others will allow the user to customize the output. The customization can range from modifying file names, to changing the order of blocks of data, to splitting the data between files, to changing the data structure, to changing from text to binary, *etc.*
Binding Time	specification or build or run time

A.5.4 System Constraints

Item Number	P20
Corresponding Variability	V20
Range of Parameters	Most operating systems that are in common use will have mesh generators that will run on the system. Some examples include: Windows, Unix, Mac OS X, Linux, *etc.*
Binding Time	specification, build or run time

A.6 References

[1]O. Zienkiewicz and D. V. Phillips. An automatic mesh generation scheme for plane and curved surfaces by 'isoparametric' co-ordinates. International Journal for Numerical Methods in Engineering, pages 519-528, 1971.

[2]P.J.Frey and P.L.George. Mesh Generation Application to Finite Elements. Hermes Science Europe Ltd, 2000.

[3]W. S. Smith and C.H. Chen. Commonality and requirements analysis for mesh generating software. In Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2004), pages 384-387, Banff, Alberta, Canada, June 2004.

A.7 Appendix: Topology Patterns for Structured Meshes

The appendix contains the possible templates for the local pattern in the element topology of a structured mesh. The list of patterns are obtained by considering the quadrilateral and triangular elements. It is obvious that there is only one possible type of quadrilateral elements as shown in "quad" in Figure 1. For triangular elements, we start with the same pattern of quadrilateral elements, and divide each quadrilateral diagonally into triangles; first in the same direction (e.g. left and right), then in different directions. This gives us $2^4 = 16$ possible patterns, of which six are symmetric as shown in Triangle1) – Triangle8) in Figure 1. Since we are only interested in symmetric patterns, these six patterns will be included. Finally, we divide each quadrilateral diagonally twice to obtain the pattern h) in Figure 1.

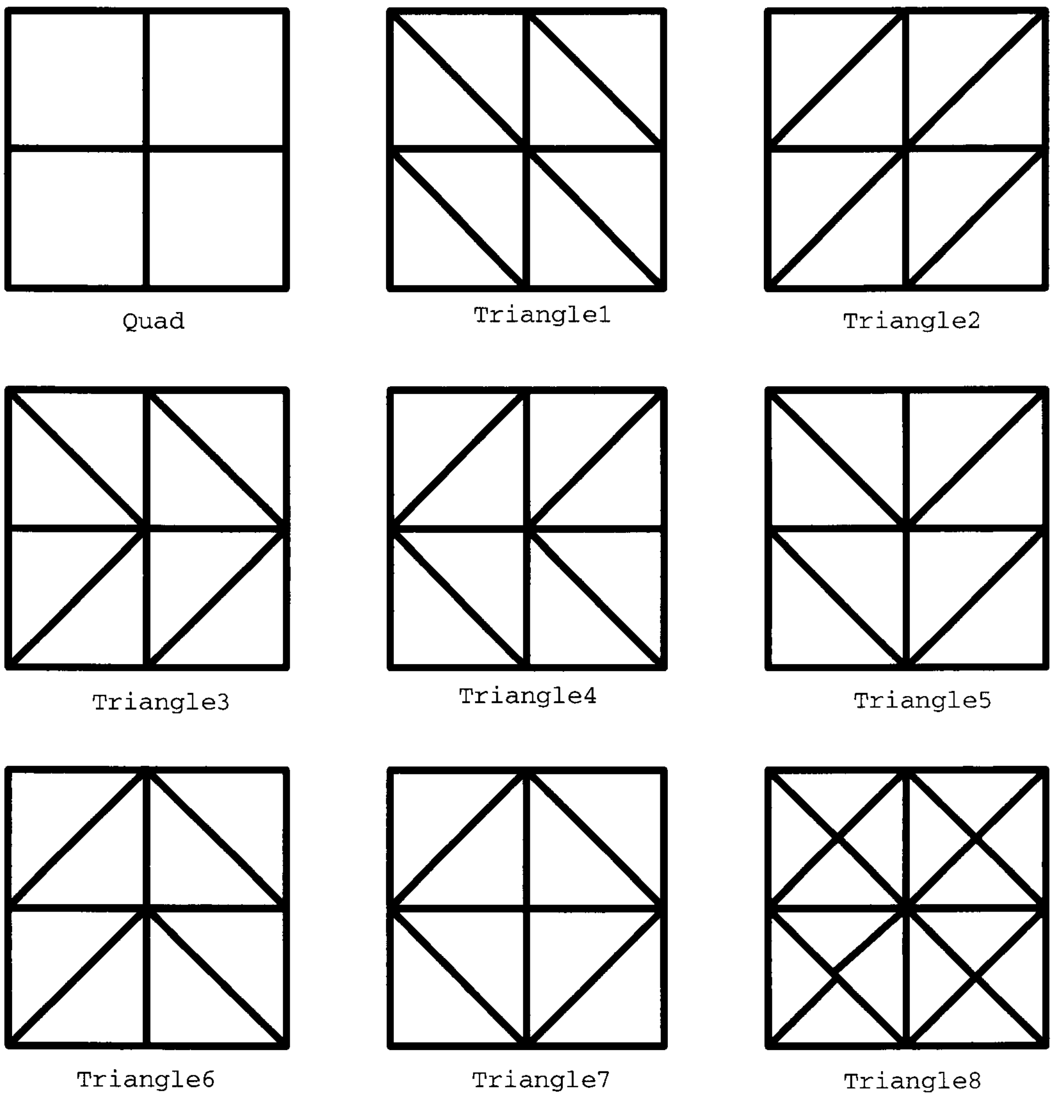


Figure A.1: Parameters of variation for patterns for structured mesh elements

Appendix B

The Software Requirement Specification for a "Parameterized Mesh Generator"

B.1 Introduction

(Content: This section contains the purpose of the document, lists of definitions and terminologies that appear later in the document, references used for the document, and an overview of the rest of the document.)

(Motivation: To provide an introduction to this document.)

To help improve the readability of this SRS document, we will start each section and its subsections with a brief discussion about its content and motivation. As part of the convention, they will be written in italics to distinguish them from the other parts of the document.

B.1.1 Purpose of Document

(Content: This subsection discusses the purpose of writing this SRS and introduces the intended audience of the SRS.)

(Motivation: To provide a basis so all readers of the SRS can have a common understanding of the intended use of this document.)

The purpose of writing this Software Requirement Specification (SRS) is to define the requirements of our system " Parameterized Mesh Generator" (PMG), and provide a "black-box" description in terms of its performance and interaction with the external environment. The intended audience of this SRS includes system users such as mesh generator builders and mesh generator users, whose roles will be defined in Section 1.2.1. Moreover, the developing personnel can refer to this document for design verification as well as future maintenance and evolution.

B.1.2 Terminology and Definitions

(Contents: This section includes all the definitions for the potentially unclear terms appearing in this document.)

(Motivation: To reduce ambiguity in the document.)

This section contains two subsections. The first subsection explains the definitions used in the software engineering field. Some definitions about the " 'mesh generator' generator" are also listed in this subsection because they are related to software requirements issues as potential users of our system. These definitions are mesh generator builders and mesh generator users. The second subsection presents the definitions used in mesh generation. The definitions are ordered alphabetically; however, they are not meant to be read sequentially, but are rather intended for reference purposes. Some of the terms have already been defined in the associated commonality analysis report [48]; therefore, we will not redefine them here, but rather refer to the previous document to provide their definitions.

B.1.2.1 Software Engineering Related Definitions and Acronyms

(Contents: This section includes the definitions for the potentially unclear terms from the software engineering field.)
(Motivation: To reduce ambiguity in the document.)

Commonality Please refer to [48].

Constraint A design constraint is an imposing factor that may limit or modify the design options. Unlike functional requirements of the system, a constraint expresses "how" the system should be implemented, instead of "what" the system should accomplish.

Context The interaction between the system to be built and the environment, which consists of the people, other systems, and technology that interfaces with the system.

Domain-specific Language (DSL) A programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to a particular problem domain [14].

Mesh Generator Builder (MGB) The person(s) responsible for generating members in our mesh generator program family. They are required to be familiar with our DSL. Their task is to work with mesh generator users to gather mesh inputs such as application domain, geometry description and physical attributes, mesh output file formats, etc. The mesh generator builder records these mesh inputs in the DSL and use the PMG to generate mesh generators. The mesh generator builder and mesh generator user can be the same person.

MG Mesh Generator

PMG "Parameterized Mesh Generator"

Mesh Generator User (MGU) The person(s) for whom the mesh generator is to be built. They will have a set of specific requirements about the type of mesh generators they expect, which will be recorded as a specification program in the DSL by the mesh generator builders to produce the corresponding mesh generator to solve their meshing problems.

Program Family Please refer to [48].

Requirement Please refer to [48].

Run time Binding Variabilities have a run time binding if the values of their parameter of variation can be fixed after the mesh generator has been built. These run time values are specified by the MGUs via the MG user interface.

Scope time Binding During design and implementation of a program family generator, it may be necessary to restrict the scope of some of the existing variabilities in the commonality analysis document. When these decisions are made, they are considered as scope time decisions. Instead of being variabilities, they will be considered as commonalities of the newly restricted program family.

Specification time Binding Variabilities have a specification time binding if the values of their parameters of variation have to be fixed before building the family member. In particular, the values are recorded by the MGBs in the specification using the DSL. Once the mesh generator is built, these values cannot be changed at run time.

Variability Please refer to [48].

B.1.2.2 Mesh Generation Related Definitions and Acronyms

(Contents: This section includes the definitions for the potentially unclear terms from the mesh generation field.)

(Motivation: To reduce ambiguity in the document.)

1D One Dimensional.

2D Two Dimensional.

3D Three Dimensional.

Cell Please refer to [48].

Conformal Mesh Please refer to [48].

Connectivity Please refer to [48].

Degree of Freedom (dof) Please refer to [48].

Domain Please refer to [48].

Element Please refer to [48].

Hybrid mesh Please refer to [48].

Mesh Please refer to [48].

Mesh Generation Please refer to [48].

Mixed mesh Please refer to [48].

Node Please refer to [48].

Physical Attribute Please refer to [48].

Structured mesh Please refer to [48].

Topology Please refer to [48].

Vertices Please refer to [48].

B.1.3 References

(Contents: This section gives the complete list of all documents referenced in the SRS.)

(Motivation: To specify the source from where the referred documents are obtained.)

[1] Fang Cao, Software Requirement Specification for " 'Mesh Generator' Generator", Appendix of Master Thesis, Computing and Software Department, McMaster University. 2006.

[2] David M. Weiss. Defining families: The commonality analysis. Submitted to IEEE Transactions on Software Engineering, 1997. URL http://www.research.avayalabs.com/user/weiss/Pub

[3] David A. Cuka and David M. Weiss. Specifying executable commands: An example of fast domain engineering. Submitted to IEEE Transactions on Software Engineering, 1997. URL http://www.research.avayalabs.com/user/weiss/Publications. html.

[4] IEEE Std. 1233, 1998 Edition, IEEE Guide for Developing System Requirements Specifications, ISBN 0-7381-1515-0 SS94659

[5] Arie van Deursen, Paul Klint, Joost Visser, Domain-Specific Languages: An Annotated Bibliography, 1998. URL: http://homepages.cwi.nl/ arie/papers/dslbib/

[6] CANSAFE / VISROCK - A Windows Application Package For Stress Analysis Using Finite Element Techniques, The Numerical Modeling Group Mining Research Laboratories, Natural Resources of Canada, Division Report: MRL 94-046(TR), October 1994

[7] Grid Generator, Dr. Stolle, Department of Civil Engineering, McMaster University

[8] C.Geuzaine and J.Remacle, Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities, March 2005. URL:http://www.geuz.org/gmsh/

[9] S.Smith and C.H.Chen, Commonality Analysis for Mesh Generating Systems, Technical Report CAS-04-10-SS, Computing and Software Department, McMaster University. 2004

B.1.4 Overview

(Contents: This subsection describes what the rest of the SRS contains.)

(Motivation: To introduce how the SRS is organized.)

The remainder of this SRS document explains the concept of the PMG and defines the specific requirements of the PMG.

Section 2 provides the overall description of the system. It does not provide specific requirements, but rather it provides background information to make the requirements easier to understand. Section 3 lists the specific requirements for our system, separated into functional requirements, non-functional requirements, and system constraints. Section 4, the last section of this SRS, discusses some issues that arose in the course of composing and revising the document.

B.2 General System Description

(Contents: This section contains general information about the PMG, including the system purpose, scope, context, major capabilities, user characteristics, and assumptions.)

(Motivation: To provide a background for the system requirements.)

B.2.1 System Purpose

(Contents: This section contains a description of current methods of mesh generator development that motivates the development of the PMG.)

(Motivation: To provide a typical scenario for mesh generation and the reasons for developing PMG.)

Our system, which is named the "Parameterized Mesh Generator" or PMG, represents an attempt to apply the idea of program family design into mesh generating systems, also known as mesh generators. A mesh is a discretization of a geometric domain into small simple shapes, such as line segments in 1D, triangles or quadrilaterals in 2D, and tetrahedral or hexahedra in 3D. Meshes are popular in many application areas such as geography and cartography. The principal application of interest in the current context is the finite element method, where meshes are essential in the numerical solution of partial differential equations. Mesh generation by hand is too demanding, especially when the computational domain is complex or the mesh contains too many elements. We need mesh generators to automatically generate meshes for us. When using a mesh generator, the user only needs to concentrate on a few input parameters and relies on the mesh generator to produce the correct mesh.

Based on the degree of generality, mesh generators today can be classified into two broad categories: general purpose and special purpose. General-purpose mesh generators are intended to solve a wide range of problems. They usually support a wide selection of element types in arbitrary domain shapes in one, two or three dimensional space, and offer flexible options via a variety of mesh generation tools. While these characteristics of general purpose MGs are appealing, their versatility implies that they are not ideal for problems in specific domains. For example, if an engineer faces problems that involve repeatedly solving rectangular domains, a general purpose mesh generator that supports arbitrary closed shape is more complex than he needs. Working with the general purpose interface will be more time consuming than necessary, and it may even distract the engineer from his original problems. Moreover, the wide variety of options available in general purpose MGs makes them difficult to learn and use. Special-purpose mesh generators are tailored for the specific physics of the problem of interest. Besides having a simpler interface for data input, they are superior to general purpose MGs for specializing output file formats because the general purpose MGs may provide an overwhelming number of choices, or possibly use a proprietary data format.

Given the advantages of special purpose mesh generators, it would be very useful to develop them systematically. Our research goal is to study the commonalities and variabilities between special purpose mesh generators and design a system for rapid development of special purpose MGs.

Our research shows that it is advantageous to develop MGs as a program family. The suitability of applying program family design into mesh generators has been argued in Smith and Chen [48] by showing that mesh generators meet the three hypotheses of a program family proposed by Weiss [53]. The idea of program families in software design has been considered by Weiss in the context of FAST (Family oriented Abstraction, Specification and Translation) [13]. The idea of " 'mesh generator' generator" centers on generating a set of special purpose MGs. These MGs form a program family by sharing the commonalities listed in ref.[48] as their common requirements, and their differences are set by assigning different values to the parameter of variations, thus fixing the uniqueness of each family member. The commonality analysis performed in ref. [48] serves as a starting point for designing our DSL. Once the DSL is developed, the family members can be rapidly generated using the language. The DSL must be developed to allow one to specify the values for all parameters of variation, which can be fixed either at specification time in the DSL or can be postponed until run time. In the proposed DSL, the choice of postponing the specification until run time would be reflected by the absence of specifying the parameter of variation in the DSL specification.

B.2.2 System Scope

(Contents: This subsection provides a short description of what the system will and will not do.)

(Motivation: To define the scope of our system.)

During our commonality analysis, some design decisions were made at scope time. We list them briefly here as our system scopes. The requirements that are also scope time decisions will be clearly indicated. For more details, please refer to the section on specific system requirements. In addition, the term "mesh generator" used in the list refers to all potential mesh generators in our program family.

- All mesh generators only generate structured meshes with a 2D topology and a 2D space.
- The mesh generators do not incorporate mesh optimization features such as smoothening and refinement/coarsening.
- The vertices and nodes are ordered counter-clockwise by convention.
- The values of mesh quality attributes, such as aspect ratio, will not be explicitly specified by the program user.
- The mesh generators will not produce the following meshes: hybrid, non-conformal or mixed.
- The type of coordinate system to describe the boundary geometry is always a Cartesian coordinate system.
- All mesh generators provide a graphical user interface to display the mesh on the screen.

- All mesh generators provide the same interface for specifying the domain.
- It is not possible to set boundary conditions so that two or more degrees of freedom (dof) have the same value.
- The mesh generators will not allow specification of internal boundaries.
- All mesh generators will write mesh information into text format.
- The tolerance and precision for each mesh output will not be explicitly controlled by the user.
- The PMG and the family member MGs should satisfy the nonfunctional requirements in accord with standard engineering analysis software. These nonfunctional requirements are usability, portability, maintainability, performance, and robustness.

B.2.3 System Context

(Contents: This subsection includes diagrams and narrative that defines all the important interfaces across system boundaries.)
(Motivation: To provide the context of the system.)

Figure 1 shows the context for our system. A rounded rectangle represents an external entity, and a rectangle is the system itself. Arrows represents the data flows between entities.

Figure 1 shows that the mesh generator user (MGU) works with the mesh generator builder (MGB) to determine the mesh-related inputs. During the preparation of the mesh inputs, it is important to understand the mesh generator user can be the same person as the mesh generator builder. Also, the mesh generator user is not necessarily one person; the term could refer to a group of people who need to solve the same types of meshing

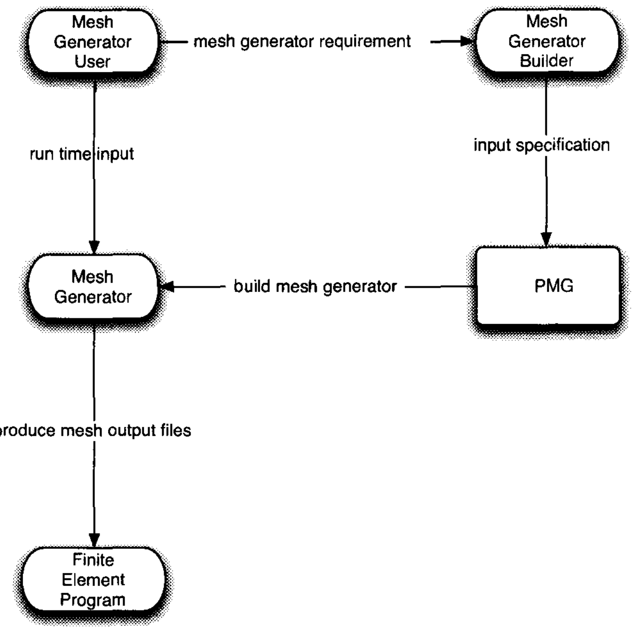


Figure B.1: System Context

problems. The mesh generator builder must translate these mesh inputs into a requirement specification using our DSL. The PMG then reads the specification and generates a special purpose MG family member. The PMG can be used repeatedly until all the necessary MGs are built. If the requirements specification for an MG should change in the future, the PMG can be run again to produce a new family member that meets the new requirements. Mesh generators are the output of PMG and they are considered as external entities with respect to PMG. Each MG produced by the PMG is a member of the program family. The MGUs will use the generated MGs to solve their meshing problems.

B.2.4 Major System Capabilities

(Contents: Diagrams and narrative will be included in this subsection to show the fundamental features of the systems needed by the user.)

(Motivation: To give an overview of the essential features of the system as well as the relationship between them.)

Since the mesh generators built by PMG are external entities, we will present the capabilities of MG and PMG in separate diagrams. Figure 2 shows the major capabilities of PMG and how they collaborate in the system. The squares, diamonds, and rounded rectangles denote system processes, decisions, and start/stop actions respectively. Parallelograms represent input/output. The rectangle with two extra vertical bars denotes an external entity or process. There are two types of arrows. A black arrow denotes a sequence flow between internal entities within the current system scope. A dashed arrow denotes a sequence flow from internal entities to external entities. Figure 3 shows the mesh generation process as performed by an MG.

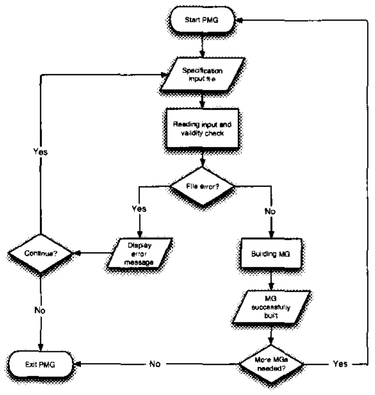


Figure B.2: Flow chart of PMG capabilities

The major capabilities of PMG includes the following: Upon the request of MGB, PMG opens and asks for the location of the input specification. If the file is found, PMG will read its content and perform a validity check. If the specification cannot be found or it is syntactically incorrect, the system will halt and display an error message. The error message will prompt the MGB to specify the correct DSL specification, and perform the same validity check again. If the specified DSL specification is correct, the PMG starts

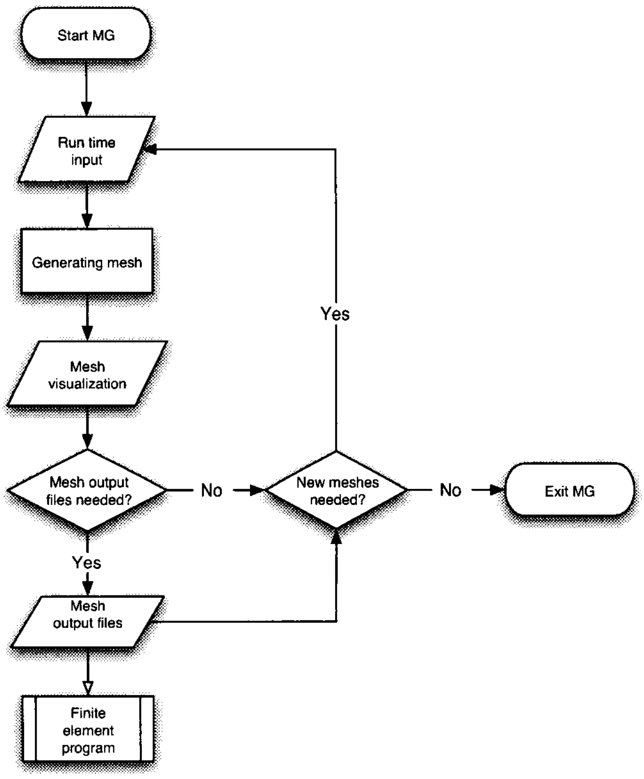


Figure B.3: Flow chart of MG capabilities

the building process, which upon successful completion, will produce a mesh generator. At this moment, the MGB can choose to either restart the PMG if there are more mesh generators to build or terminate the program.

The process now shifts to mesh generation. As a scope time decision, all MGs generated by the PMG will provide a graphical user interface. If there are mesh inputs that need to be specified at run time, the MGU will specify them in the MG interface and generate the mesh. After the mesh visualization is displayed on the screen, the MGU should decide whether he needs the output files. If the output files are needed, they will be generated in the format specified by the MGU. The principal use of the files is that they are the inputs of external finite element programs. The MG can be used repeatedly if the MGU needs a new mesh and the requirements stay the same.

B.2.5 User Characteristics

(Contents: This section reviews the potential users of the system and their characteristics.)
(Motivation: To give consideration on the qualification of the system users and make sure they are consistent with the required knowledge level.)

The target user group of the PMG includes engineers, project managers, students, and other possible users involving in finite element analysis and requiring relatively easy and fast special purpose mesh generators. They are expected to have completed first year engineering or science at the university level or equivalent. They should also be knowledgable in numerical methods, and know how to use Windows operating system.

B.2.6 Assumptions and Dependencies

(Contents: This section lists the factors that affect the requirements stated in the SRS.)
(Motivation: To make everyone aware of the assumptions that are made in this SRS.)

By the time the system is to be released, personal computers are existing and widely used.

B.3 Specific System Requirements

(Contents: This section contains all the system requirements. The types of requirements listed here are functional requirements, non-functional requirements, and system constraints.)
(Motivation: To list every requirement of the system in one section.)

The requirements listed in this SRS document are formulated based on the commonality analysis discussed in ref. [48]. The commonalities and variabilities listed in ref. [48] provides an excellent groundwork for the requirements. The commonalities represent the uniform properties of all the mesh generators in the context of the commonality analysis; therefore, they should be requirements in our SRS. The variabilities reflects the distinction between family members, and they should also be requirements. However, the distinction of family members is shown by the parameters of variation and their binding time; therefore, we include the parameters of variation and binding time information with the requirements associated with variabilities. To ensure that this SRS document is complete, we can check whether all the commonalities and variabilities have been referred to by some requirements in the SRS. A systematic approach to do this is through a table similar to traceability matrix. The table that we made to check our SRS completeness is included in Appendix B.4.

All the system requirements are presented in a table format. Each requirement has a unique requirement number, starting with capital "R". Each requirement is given in a separate table with the following rows: requirement ID, description, commonality reference, parameter of variation, binding time, and history. Rows three to five are used to cross-reference the requirements to the commonality analysis report [48], and these fields are only provided in the cases where the reference exists. The parameter of variation and binding time entries have the value '-' if the requirement is not linked with a variability. Because of the scope time binding, some requirements, for example R2, that were originally associated with variabilities are now associated with commonalities. These requirements will have the value '-' as the parameter of variation, and have the value 'scope time' as the binding time.

B.3.1 Functional Requirements

(Contents: This section contains the specification of each individual functional requirement of PMG.)

(Motivation: To list the functional requirements of the system.)

This section is divided into two subsections: PMG features and input. The section of PMG features lists the requirements of PMG to generate a correct mesh generator. These requirements must be captured to describe the core functionalities of PMG. The section on input presents the expected contents of mesh inputs, such as geometry description and output file formats, that are read by the PMG in the DSL. As discussed earlier, the values of these inputs can be specified either via our DSL at specification time or via the graphical user interface at run time. It is worth noting that the term "mesh generator" used in the context refers to all mesh generators in our program family.

B.3.1.1 PMG features

(Contents: This section contains the functional requirements of PMG.)

(Motivation: To describe the functionality of PMG.)

Requirement ID	R1

Description	After PMG reads the DSL input specification, it should either produce a mesh generator if there is no error in the DSL specification, or halt.
Commonality reference	–
Parameter of variation	–
Binding time	–

Requirement ID	R2
Description	The system PMG should provide the functionality to incorporate a graphical user interface for all family members.
Commonality reference	V21
Parameter of variation	All mesh generators use a graphical user interface.
Binding time	scope time

Requirement ID	R3
Description	During the code generation process of PMG, it should distinguish the set of parameters of variation specified at specification time and at run time. If a parameter of variation is specified in the DSL, it cannot be changed at run time. Otherwise, it must be specified in the mesh generator interface at run time by the MGUs before generating a mesh.
Commonality reference	–
Parameter of variation	–
Binding time	–

B.3.1.2 Input

(Contents: The requirements listed here relate to possible ways to specify mesh inputs and their formats. These inputs are for the PMG in order to generate a MG as a family member.)
(Motivation: To list the requirements about inputs.)

Requirement ID	R4
Description	To define a meshing problem, the MGUs should provide a DSL specification that contains the mesh inputs, such as geometry description and output file formats, to the PMG.
Commonality reference	C8
Parameter of variation	–
Binding time	–

Requirement ID	R5
Description	The mesh inputs should allow one to specify a string for the targeted problem domains corresponding to a large range of problems that can be solved via finite element analysis.
Commonality reference	V1, P1
Parameter of variation	The problem domain includes solid mechanics, fluid mechanics, heat transfer, etc.
Binding time	specification or run time

Requirement ID	R6
Description	The geometric domain of the problem specified in the mesh inputs should be a closed boundary.
Commonality reference	C9

Parameter of variation	–
Binding time	–

Requirement ID	R7
Description	The mesh inputs should provide a parametric representation to define the boundary of the computational domain.
Commonality reference	V23, P23
Parameter of variation	-
Binding time	scope time

Requirement ID	R8
Description	The mesh inputs should include the dimensionality of the geometric coordinates.
Commonality reference	V19, P19
Parameter of variation	2D
Binding time	specification or run time

Requirement ID	R9
Description	The mesh inputs should include the dimensionality of mesh elements.
Commonality reference	V19, P19
Parameter of variation	2D
Binding time	specification or run time

Requirement ID	R10

Description	The mesh inputs should include physical attributes, such as the material properties, the boundary conditions, etc.
Commonality reference	C10
Parameter of variation	–
Binding time	–

Requirement ID	R11
Description	The mesh inputs should allow one to specify the number of subdivisions along each direction for structured meshes.
Commonality reference	V22, P22
Parameter of variation	The number of subdivisions are variable depending on the shape of the computational domain, ranging from 1 to many for each boundary edge.
Binding time	specification or run time

Requirement ID	R12
Description	The mesh inputs should include options to allow the user to specify the number of material properties, their names and their types.
Commonality reference	V24, P24
Parameter of variation	The number of material properties is variable, ranging from 1 to many. The names of material properties can be any string defined by the user. The possible types of material properties can be the following: integer, real, and boolean.
Binding time	specification or run time.

Requirement ID	R13
Description	The mesh inputs should specify the number of materials allowed in the computational domain.
Commonality reference	V25, P25
Parameter of variation	The entire domain can consist of one material or there may be a finite number of different materials.
Binding time	specification or run time.

Requirement ID	R14
Description	The mesh inputs should specify boundary conditions details, such as name, type and value.
Commonality reference	V26, P26
Parameter of variation	The boundary conditions are of Neumann or Dirichlet type. If a Dirichlet boundary condition has the value zero, it will typically be specified using the fixity information.
Binding time	specification or run time

Requirement ID	R15
Description	When specifying boundary conditions, a maximum of one condition can be given for each degree of freedom. For instance, a dof cannot have both a prescribed displacement and a prescribed force.
Commonality reference	C11
Parameter of variation	–
Binding time	–

Requirement ID	R16
Description	The number and meaning of system parameters may be different. These parameters can be passed to a finite element program.
Commonality reference	V27, P27
Parameter of variation	System parameters may include the degree of implicitness for a time marching scheme. The number and type of these parameters may vary.
Binding time	specification or run time

Requirement ID	R17
Description	The mesh inputs should allow one to specify the number of nodes for an element and the location of these nodes.
Commonality reference	V10, P10
Parameter of variation	The element can have fewer nodes than vertices, the same number of nodes as vertices or more nodes than vertices. The nodes can be located at the vertices, on the element edges, or inside the element.
Binding time	specification time or run time

Requirement ID	R18
Description	The mesh inputs should include the number of dof at a node and the meaning of each of those dof.
Commonality reference	V11, P11

Parameter of variation	The number and type of degrees of freedom at the nodes can vary between different types of elements and within an element.
Binding time	specification or run time

Requirement ID	R19
Description	The mesh inputs may include the pattern of the number of dofs that vary between the nodes of an element.
Commonality reference	V12, P12
Parameter of variation	If the geometry is interpolated at fewer nodes than the interpolation of the dof, the element is called subparametric. If the geometry is interpolated at the same number of nodes as that for the dof, the element is isoparametric. If the geometry is interpolated at more nodes than that for the dof, the element is superparametric.
Binding time	specification or run time

Requirement ID	R20
Description	The mesh inputs should include the possible types of local topology patterns allowed by the computational domain.
Commonality reference	V7, P7
Parameter of variation	In 2D element topology, the possible template may be one of the one of the nine potential local topology templates provided in Appendix A.
Binding time	specification or run time

Requirement ID	R21
Description	The mesh inputs should not allow two dofs to have the same value.
Commonality reference	V28,P28
Parameter of variation	–
Binding time	scope time

Requirement ID	R22
Description	The mesh inputs should not allow specification of internal boundaries.
Commonality reference	V29, P29
Parameter of variation	–
Binding time	scope time

Requirement ID	R23
Description	The mesh inputs should specify the number of output file(s) generated by the mesh generator.
Commonality reference	V32, P32
Parameter of variation	The value for the number of output files can range from 1 to many.
Binding time	specification or run time

Requirement ID	R24
Description	The mesh inputs should provide some degree of customization of output file(s).

Commonality reference	V36, P36
Parameter of variation	The degree of customization may vary for different mesh generators. It can range from modifying file names, to changing the order of blocks of data, to splitting the data between files, etc.
Binding time	specification or run time

Requirement ID	R25
Description	The mesh inputs should allow one to specify the order of element information of a mesh.
Commonality reference	V34, P34
Parameter of variation	The element information of a mesh is either listed in an increasing order (implicitly), or in arbitrary order.
Binding time	specification time or run time

Requirement ID	R26
Description	The mesh inputs should allow one to specify the order of nodal information of a mesh.
Commonality reference	V35, P35
Parameter of variation	The nodal information of a mesh is either listed in an increasing order (implicitly), or in arbitrary order.
Binding time	specification time or run time

B.3.2 Nonfunctional Requirements

(Contents: The nonfunctional requirements are listed here. They are concerned with qualities of system performance, usability, maintainability, etc.)

(Motivation: To list the nonfunctional requirements.)

Requirement ID	R27
Description	The PMG should perform its essential functions in reasonable time, such as instantaneously for entering user input, seconds for waiting graphics display, or minutes for mesh generation and output file creation.
Commonality reference	C15, V39
Parameter of variation	–
Binding time	scope time

Requirement ID	R28
Description	The accuracy for each output is not explicitly controlled by the MGUs. The tolerance will be consistent with the usual expectations for engineering and scientific computation software.
Commonality reference	C16, V40
Parameter of variation	–
Binding time	scope time

Requirement ID	R29

Description	The precision for each output is not explicitly controlled by the MGUs. The precision will be consistent with the usual expectations for engineering and scientific computation software.
Commonality reference	C17, V41
Parameter of variation	–
Binding time	scope time

Requirement ID	R30
Description	An anonymous survey shall show that 90% of a test panel of MGUs with basic computer skills and mesh software experience can use PMG to generate a rectangular domain, and discretize it with one material property and one boundary condition on one edge within 2 minutes.
Commonality reference	–
Parameter of variation	–
Binding time	scope time

Requirement ID	R31
Description	The mesh quality parameters such as aspect ratio, minimum angle, etc. are not controlled by the MGUs. However, the quality of the meshes will be consistent with the ones that are generally created by engineering computing software.
Commonality reference	V15
Parameter of variation	–

Binding time	scope time

Requirement ID	R32
Description	The PMG should take no more than 20 minutes for an average user to learn how to discretize a simple rectangular domain with one material property and one boundary condition on one edge.
Commonality reference	–
Parameter of variation	–
Binding time	scope time

Requirement ID	R33
Description	The PMG should be developed to reduce the time spent on maintenance. The redevelopment should not take more than 25% of the time as the first development.
Commonality reference	–
Parameter of variation	–
Binding time	scope time

B.3.3 System Constraints

(Contents: The requirements for system constraints relate to the external environments.)

(Motivation: To list all the constraints imposed on the system.)

Requirement ID	R34

Description	The first version of PMG is designed to run under Windows and MacOS because of the time constraint. However, we intend to develop the future versions to run under multi-platforms such as Linux.
Commonality reference	C19, V37, P37
Parameter of variation	–
Binding time	scope time

Requirement ID	R35
Description	The system requires reasonable amount of memory and disk storage, consistent with the resources of a typical personal computer in the year 2006.
Commonality reference	V38, P38
Parameter of variation	–
Binding time	scope time

B.4 Other System Issues

(Content: This section contains some other supporting information important to PMG development. It includes open issues, off-the-shelf solutions, our program family, and waiting room.)

(Motivation: To present a more complete picture of all factors that might contribute to the PMG development.)

B.4.1 Open Issues

(Contents: A statement of factors that are uncertain and may have an impact on the system are listed here.)

(Motivation: To discuss the uncertainties appearing in PMG design and provide objective input to risk analysis.)

In the course of composing and revising the above documentation, the following issues arose:

- The documentation and definition of a mesh should be made more formal.
- The design of our DSL has many options. We can choose an existing language such as XML to implement our DSL. However, the choice of DSL may have an impact on the performance of the PMG. It may be advantageous if a new DSL is developed in the future that offers the best way to capture the mesh generator requirements.

B.4.2 Off-the-Shelf Solutions

(Contents: A list of existing systems similar to our research goal, if any, is stated here. They could be the potential solutions to our research problems.)

(Motivation: To give consideration to whether or not our solution can be bought or borrowed.)

The idea of creating a program family of mesh generators is an unexplored and challenged research area. So far, we have not been able to find an existing solution to our research problem. However, we can study the design from existing mesh generators such as CanSafe [51], Grid Generator [49], Gmesh [20], etc.

B.4.3 Our Program Family

(Content: This section provides a blueprint of how the system will be extended. It separates the requirements for the first version of the system with the potential requirements of future versions of the system.)

(Motivation: To make plan for the project management.)

This section discusses how the requirements listed in Section B.3 will be implemented in different versions of the system. All the requirements listed in section B.3 will be implemented in the first version. However, some scope time decisions can be relaxed or changed for future versions of the system. For example, a future software version can be designed to accommodate a mixed mesh, a hybrid mesh, or a nonconformal mesh. If the future program family offers more flexibilities, this SRS document needs to be revised to accommodate the new requirements.

B.4.4 Waiting Room

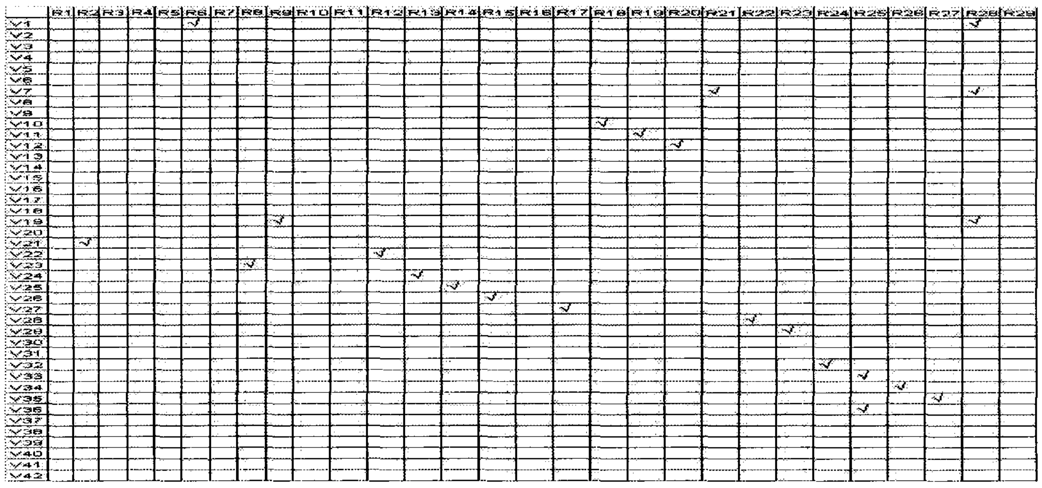
(Contents: This section contains some potential requirements of the future version of the system.)

(Motivation: To make plans for the project development.)

- The PMG could be designed for unstructured mesh generators.
- The element topology may include 3D structures such as tetrahedral or hexahedra elements.
- The MGU may specify whether or not to incorporate mesh optimization features.
- The possibility of adding complex meshes such as hybrid, mixed, or nonconformal meshes can be considered.

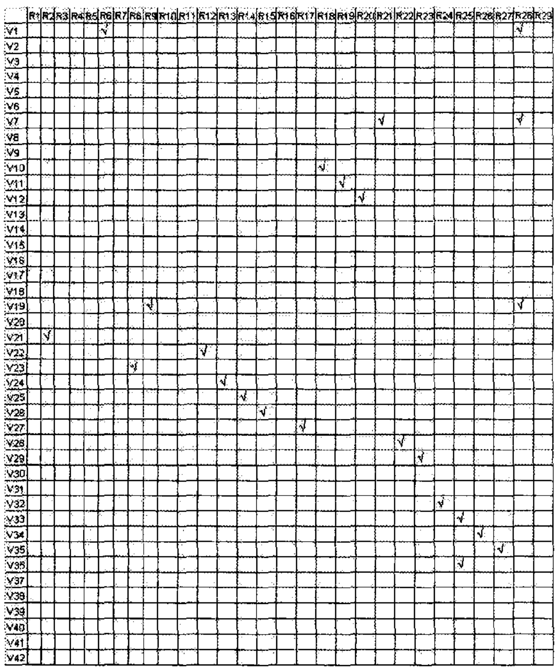
Appendix B.A

This section illustrates the tabular approach to verify the completeness of the SRS. The rows denote the commonalities and variabilities presented in the commonality analysis in reference [48]. The columns represent the requirements formulated in the SRS. A symbol '✓' is filled at the intersection of a row and a column if the commonality or variability in the row is fulfilled by the requirement in the column. A systematic check of the completeness of the SRS was done by examining whether all rows representing commonalities and variabilities contain one or more check marks.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29
V1						✓																						✓	
V2																													
V3																													
V4																													
V5																													
V6																													
V7																					✓							✓	
V8																													
V9																													
V10																		✓											
V11																			✓										
V12																				✓									
V13																													
V14																													
V15																													
V16																													
V17																													
V18																													
V19									✓																			✓	
V20																													
V21		✓																											
V22												✓																	
V23								✓																					
V24													✓																
V25														✓															
V26															✓														
V27																	✓												
V28																						✓							
V29																							✓						
V30																													
V31																													
V32																								✓					
V33																									✓				
V34																										✓			
V35																											✓		
V36																									✓				
V37																													
V38																													
V39																													
V40																													
V41																													
V42																													

Figure B.4: Requirements formulated to fulfill commonality requirements



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29
V1						√																						√	
V2																													
V3																													
V4																													
V5																													
V6																													
V7																					√							√	
V8																													
V9																													
V10																		√											
V11																			√										
V12																				√									
V13																													
V14																													
V15																													
V16																													
V17																													
V18																													
V19									√																			√	
V20																													
V21		√																											
V22												√																	
V23								√																					
V24													√																
V25														√															
V26															√														
V27																	√												
V28																						√							
V29																							√						
V30																													
V31																													
V32																								√					
V33																									√				
V34																										√			
V35																											√		
V36																									√				
V37																													
V38																													
V39																													
V40																													
V41																													
V42																													

Figure B.5: Requirements formulated to fulfill variability requirements

Appendix C

The Module Guide for “Parameterized Mesh Generator” (PMG)

C.1 Introduction

This document, known as the module guide for PMG, discusses the organization of our software based on information hiding and abstraction. We need to provide a structural description of our system that shows how the system is decomposed into parts and the relations between those parts. Such recent practices have been suggested to be highly useful for software documentation and future change or maintenance. A clear modular structure is intended to help the designers and maintainers identify the parts of the software they need to understand, without being confused about details of the other parts of the software. It has been argued in [8]A clear module guide would benefit further software development by achieving the following:

– As a Guide for New Project Members - The responsibilities of modules are clearly specified in the module guide. Furthermore, the hierarchical structure of the document allows the new members to quickly understand the system and find the specific module(s) they are looking for.

- As the Support for Maintainers - As mentioned above, the likely changes are made in individual modules rather than the whole system. This module guide helps the maintainers' understanding when making these changes. However, after the changes, the maintainers have to update the relevant sections of the document to reflect the new design.
- As a Verification Check for Reviewers - The module guide can be used to check for possible errors in the current design. First of all, inconsistency (e.g. duplication or gaps) among modules can be disovered. Secondly, feasibility of the decomposition can now go under investigation. Finally, the flexibility of the design can now be evaluated.

Our goal of the decomposition is to reduce the cost of software development by assigning each part of the system behaviors and functionalities into modules, and allow modules to be designed and revised independently. Therefore, we followed the principle listed as follows:

- Each module's structure should be simple enough that it can be understood fully.
- It should be possible to change the implementation of one module without knowledge of the details of other modules and without affecting the behavior of other modules.
- It should be possible to make likely changes without changing the module interfaces. Only very unlikely changes should require changes in the interfaces of widely used modules.
- It should be possible to make a major software change as a set of independent changes to individual modules. If the interfaces of the modules are not changed, it should be possible to run and test any combination of old and new module versions.

In the module description section, some of the modules have no prefixes in this document because they are assumed to be supported by the operating system or the software and hardware environment in which the PMG will operate. Therefore, these modules will not be directly implemented by us.

The rest of the document is organized into three sections: Section 2 describes all the modules in PMG. Section 3 summarizes the module hierarchy of all modules introduced in Section 2. Section 4 discusses the relationship between modules with the "use" hierarchy.

C.2 Module Decomposition

C.2.1 Hardware-Hiding module

Secrets: The data structures and algorithms used to implement the virtual hardware.

Services: This module provides the interface between the hardware and software. A typical example is to display the resulting mesh on the screen, and to accept inputs from input devices.

Prefix: -

C.2.1.1 Input Device module

Secrets: The data structures and algorithms used to implement the hardware that accepts user inputs.

Services: Serves as an interface between the system and input devices such as keyboards and mouse to receive user inputs.

Prefix: -

2.1.1.1 Keyboard Input module

Secrets: The data structures and algorithms used to implement the interface between the keyboard and the software.

Services: Receives the input from keyboard and communicates the information with the rest of the system.

Prefix: -

2.1.1.2 Mouse Motion module

Secrets: The data structures and algorithms used to implement the interface between the mouse and the software.

Services: Keeps track of mouse motion and behaviors, and communicates the information with the rest of the system.

Prefix: -

C.2.1.2 Output Device module

Secrets: The data structures and algorithms used to implement the hardware that handles system output.

Services: Serves as an interface between the system and output devices such as monitors and output files to display the system results.

Prefix: -

2.1.2.1 Screen Display module

Secrets: The data structures and algorithms used to implement the interface between the system and the monitor to display results on the screen.

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

Prefix: -

C.2.1.3 File Input/Output module

Secrets: The data structures and algorithms used to read file(s) into the software, and write text to file(s).

Services: Implement the hardware aspect of file Input/Output to read and write information between the software and file(s).

Prefix: -

C.2.2 Behavior-Hiding module

Secrets: The implementation details of the required behaviors.

Services: The behavior-hiding module contains many lower level programs that implements the requirements defined in the software requirement specification (SRS). These programs combine to provide externally visible behaviors of the system. This module serves as the communication layer between the hardware module and software decision module. The programs in this module reflect the visible functions of the implementation and they will need to be changed if there are changes in the relevant sections in the SRS.

Prefix: -

C.2.2.1 Function Drivers module

Secrets: The rules that determine the value of the inputs/outputs.

Services: Consists of a set of individual submodules called 'function drivers', and each function driver controls a set of closely related outputs.

Prefix: -

2.2.1.1 Master Control module

Secrets: How to use programs provided by other modules to start and maintain the proper sequence of programs being called.

Services: Provides the main program that controls the flow of software execution. This module acts as a central hub or a mediator among the function drivers of the system.

Prefix: **mc**

2.2.1.2 Frame Display module

Secrets: The data structure and algorithms to decide how the graphical user interface (GUI) of the system is drawn on the screen.

Services: Displays various visual components (e.g. buttons, menus, dialogs, icons, etc) that form the overall system GUI.

Prefix: -

Mesh Generator Interface module

Secrets: How the interface for a specific mesh generator family member will appear and how the user can interact with this interface.

Services: Provides the interface so that the user can generate a mesh. This module uses many programs that specify all the required information to generate a mesh.

Prefix: mgi

Geometry Specification module

Secrets: How the interface for boundary specification will appear and how the user can interact with this part of the interface.

Services: Provides the interface to let user specify the boundary information, such as the number of boundary vertices and their coordinates.

Prefix: gs

Element Specification module

Secrets: How the interface for element type specification will appear and how the user can interact with this part of the interface.

Services: Allow the user to specify the desired mesh element type. The relevant information include the span information (numbers of rows and columns), element shape and topology, coordinates of the nodes used for geometry interpolation and degrees of freedom (d.o.f), and the subdivisions along each row and column.

Prefix: es

Physical Attributes Specification module

Secrets: How this part of user interface will appear and how the user will interact with this part of the system.

Services: Lets the user specify the physical attributes of the input information.

Prefix: –

Material Property Specification module

Secrets: How the material properties are to be specified, for example, what the user interface for the specification should be.

Services: Lets the user specify the material properties of the input domain.

Prefix: mps

Boundary Condition Specification module

Secrets: How the boundary conditions are to be specified, for example, the steps taken to specify certain types of boundary conditions, how the user interface for this specification looks like.

Services: Lets the user specify the boundary conditions of the input domain.

Prefix: bcs

System Parameter Specification module

Secrets: How the interface for system parameters will appear and how the user can interact with this part of the interface.

Services: Lets the user specify the name, type, and value of the system parameter(s), if there's any.

Prefix: sps

File Output Specification module

Secrets: How the interface for output file specification will appear and how the user can interact with this part of the interface.

Services: Lets the user specify the extensible stylesheet language (XSL) files to transform the mesh information into user-customized format.

prefix: fos

C.2.2.2 Shared Services module

Secrets: The aspect of the behavior that applies to two or more of the outputs.

Services: Includes modules that are shared by two or more function drivers. If there is a change in this module, it will affect all the modules that share it.

Prefix: –

C.2.2.2.1 Error Handle module Secrets: The error detection mechanisms on handling different errors.

Services: Provides programs to handle different types of errors. (e.g. display error message, terminate the program, and etc.)

Prefix: eh

C.2.2.2.2 Mesh Drawing module Secrets: The mechanism of how to draw the mesh on the screen.

Services: Provides programs to draw the mesh on the screen.

Prefix: md

C.2.2.2.3 File Customization module Secrets: The mechanism of how to write mesh data to text files in a specified format.

Services: Provides programs to write mesh data to text files in a specified format.

Prefix: fc

C.2.3 Software Decision module

Secrets: This module contains hidden software design decisions. These design decisions may be based on mathematical theorems, physical facts, or programming considerations such as algorithmic efficiency. The secrets of this module are not described in the SRS.

Services: Receives user inputs and implement the core system functions using data types and algorithms determined by the software designers. The programs in this module do not provide direct interaction with user.

Prefix: –

C.2.3.1 Specification Parsing module

Secrets: The data structures and algorithms used to transform XML specification into a mesh generator.

Services: Parses the information from XML specification, and generate a mesh generator according to the specification.

Prefix: xsp

C.2.3.2 Input Data module

Secrets: How the user inputs are stored.

Services: Stores the user inputs in proper data structures. The inputs include the information provided in the XML specification and in the user interface at run-time.

Prefix: id

C.2.3.3 Mesh Data module

Secrets: How the mesh information is stored.

Services: Stores the complete mesh information in proper forms.

Prefix: md

C.2.3.3.1 Geometric Grid module Secrets: How the geometric aspect of mesh information is stored.

Services: Stores the geometric information of a mesh (e.g. nodal coordinates), and provides ways to import and export the information.

Prefix: gg

C.2.3.3.2 Physical Attributes module Secrets: How the physical attributes of mesh information is stored.

Services: Stores the physical attributes associated with mesh elements, and provides ways to import and export this information.

Prefix: pa

C.2.3.3.3 System Parameters module Secrets: How the system parameters of the mesh is stored.

Services: Stores the system parameters associated with mesh, and provides ways to import and export this information.

Prefix: sp

C.2.3.4 Mesh Generation Algorithm module

Secrets: The data structure and procedures used to implement the mesh generation algorithm.

Services: Implements the mesh generation algorithm to generate a mesh.

prefix: mga

C.3 Module Hierarchy

Level 1	Level 2	Level 3	Level 4
Hardware-Hiding module	Input Device Module	Keyboard Input module	
		Mouse Motion module	
		File Reading module	
	Output Device module	Screen Display module	
		File Writing module	
Behavior-Hiding module	Function Drivers module	Master Control module	
		Frame Display module	Mesh Generator Interface module
			Geometry Specification module
			Element Specification module
			Physical Attributes Specification module
			Material Property Specification Module
			Boundary Condition Specification module
			System Parameter Specification module
			File Output Specification module
	Shared Services module	Error Handle module	
		Mesh Drawing module	
		File Customization module	
Software Decision module	XML Specification Parsing module		
	Input Data module		
	Mesh Data module	Geometric Grid module	
		Physical Attributes module	
		System Parameters module	
	Mesh Generation Algorithm module		

Figure C.1: Module Hierarchy

C.4 Use Hierarchy between Modules

In this section, we include the use hierarchy between modules for the major features of the system.

The use hierarchy presented here is the "uses relation" defined by Parnas. If module A uses module B, A uses the functionalities provided by B and the correctness of A depends on the correctness of B. In the diagram, each circle represents the module that will be implemented in PMG, and the rectangles represent the modules that will not be implemented. The use hierarchy can be found as the one constructed by the arrows with filled ends, whereas the arrows with empty ends means data flow.

To aid the readers' understanding about the interactions between modules of the system, we divide the entire system into subsystems, and each of the subsystem performs a major functionality in the system. The major subsystems are PMG, boundary specification, material specification, boundary condition, mesh generation, and file format.

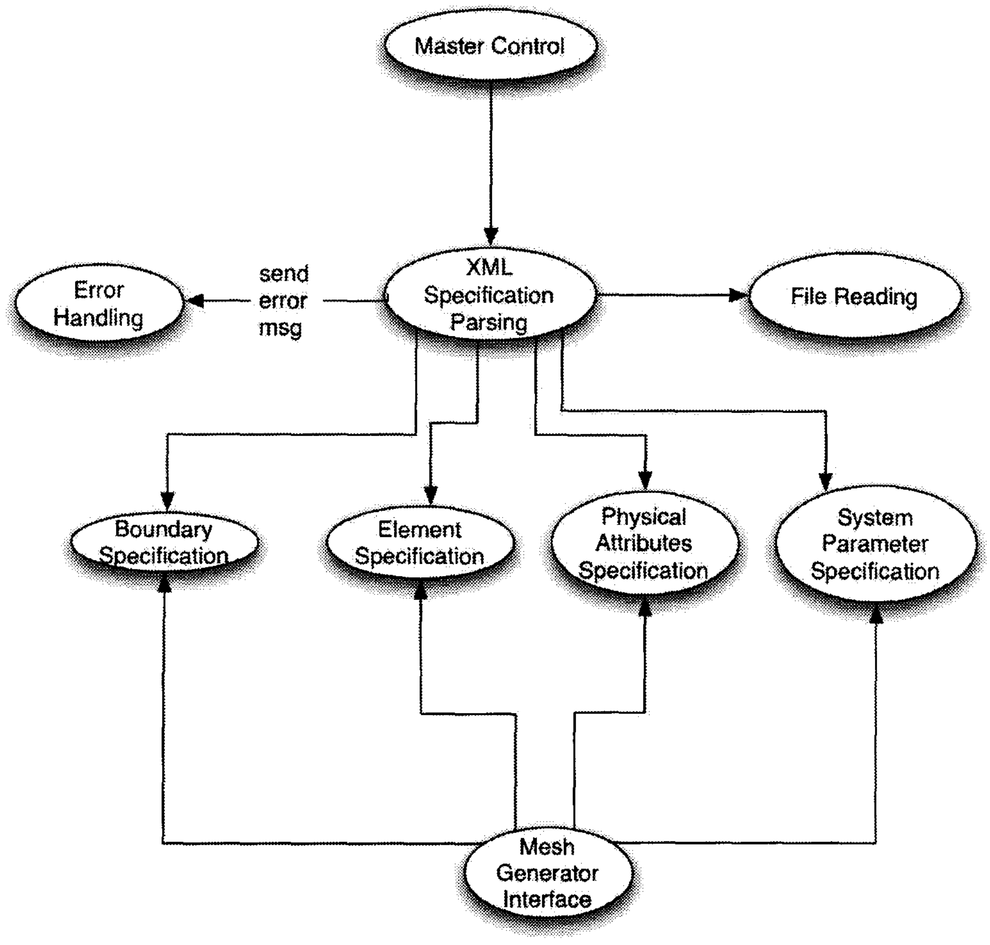


Figure C.2: Generation of mesh generators

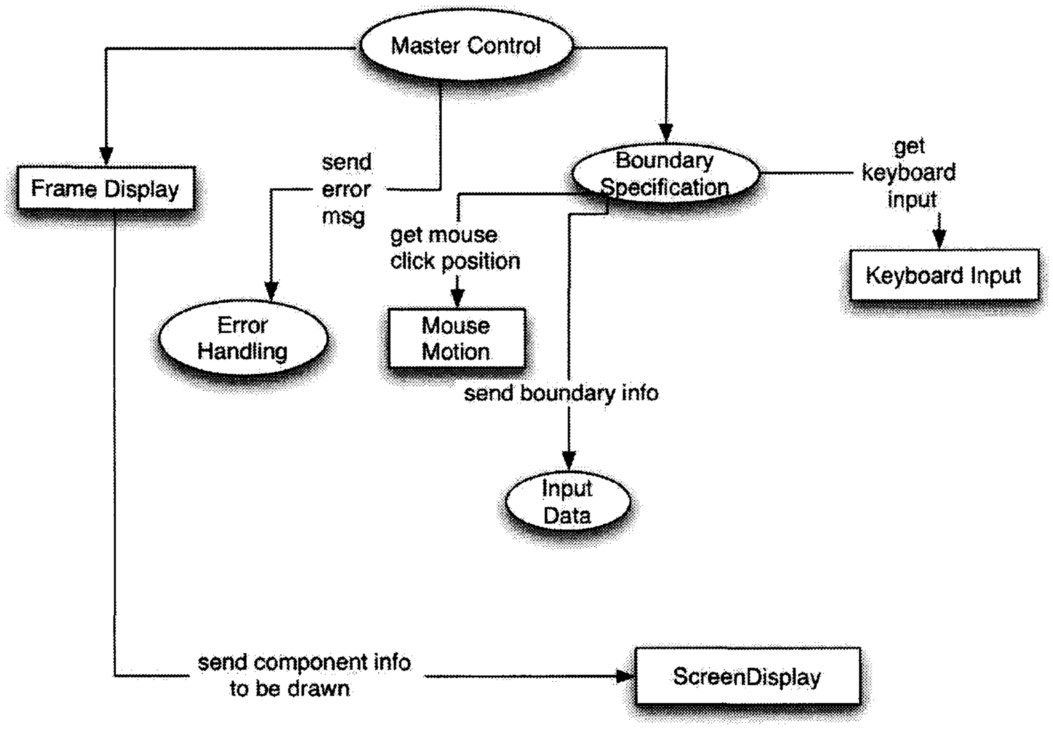


Figure C.3: Boundary Specification

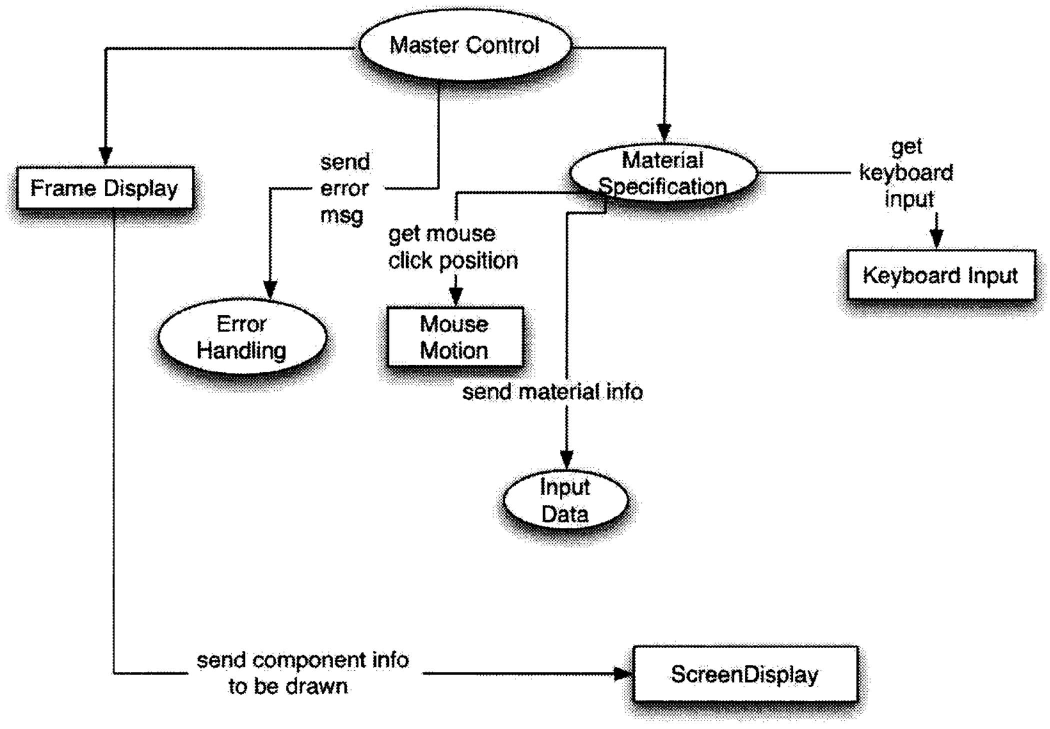


Figure C.4: Material Specification

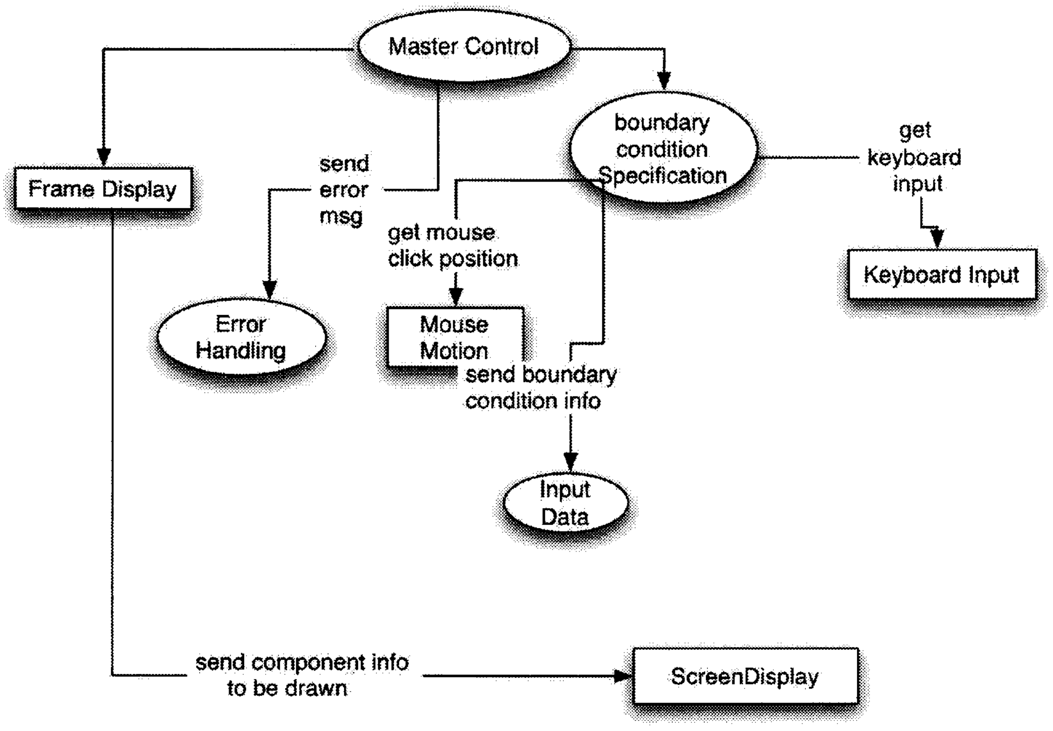


Figure C.5: Boundary Condition Specification

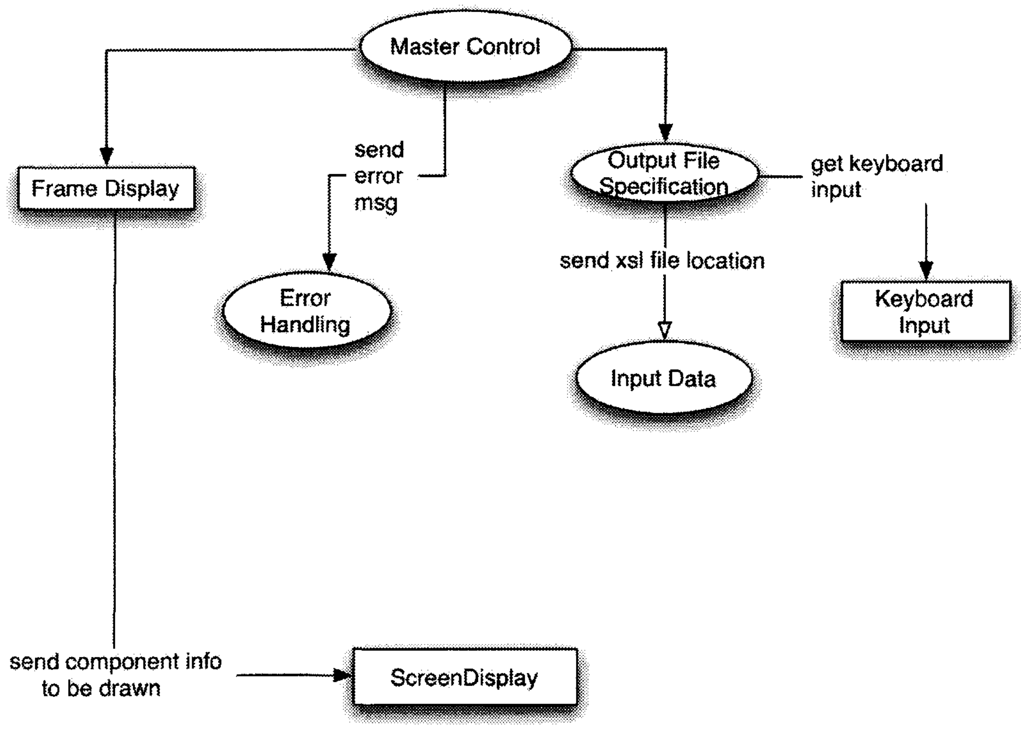


Figure C.6: File Format Specification

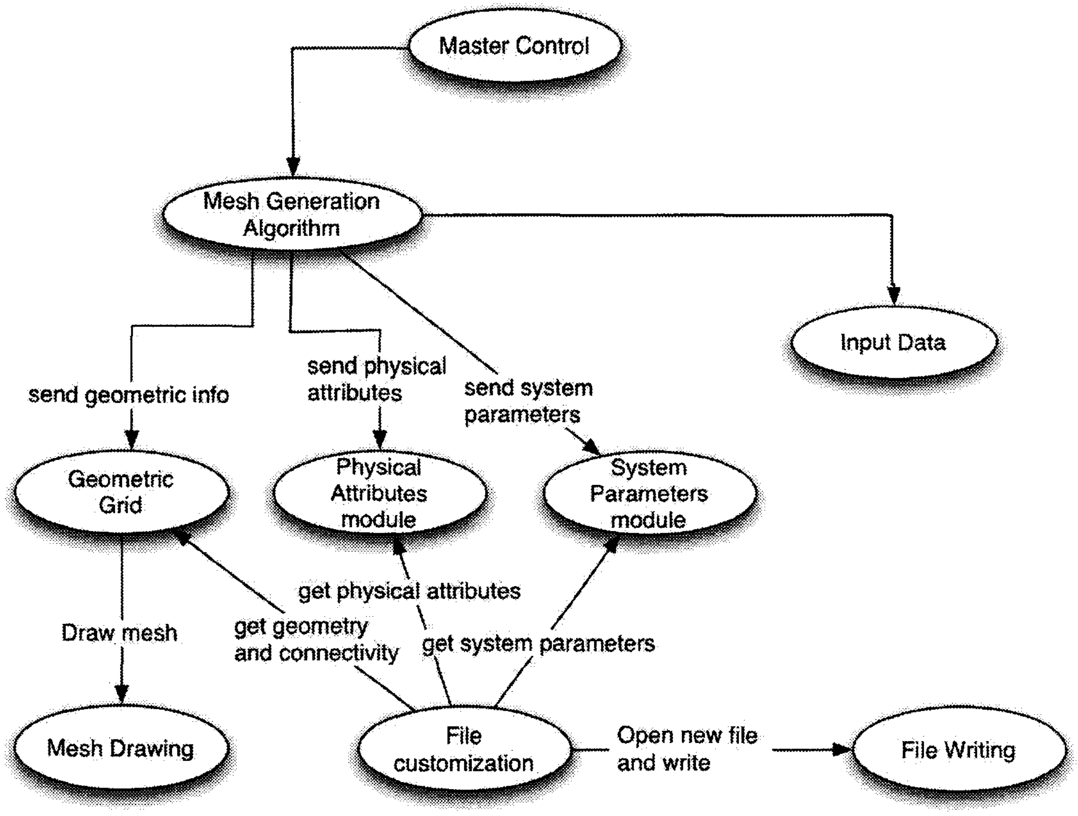


Figure C.7: Mesh Generation

Appendix D

Test Plan

D.1 General Information

The test plan describes how testing of our system will be accomplished. A good test plan documentation provides three major benefits. First, a test plan facilitates the technical tasks of testing. Test plans help improve testing coverage and efficiency. Second, a test plan improves the communication about testing tasks and process. A well-written test plan improves the communication between the programmers and the testers by providing the thinking behind the tester's strategy. Finally, a test plan provides structure for organizing and managing the testing project. If the testing is performed by a team, writing a test plan helps identify and coordinate the procedures involved in the overall testing process.

This section contains two subsections. Section D.1.1 briefly summarizes the functions of PMG. Section D.1.2 states the objectives to be accomplished by testing. Section D.1.3 presents the scope of the testing process.

D.1.1 Summary

Our system, PMG, is an attempt to systematically develop special-purpose mesh generators using the program family methodology. The system centers on using an XML specification, which describes values bound at specification time according to the definition in the commonality analysis document as in Appendix A. Given the set of variabilities, a user with certain degree of mesh generation knowledge should be able to use PMG to quickly develop a special-purpose mesh generator to solve his specialized problems. The mesh generators generated by PMG also offers capabilities to customize output files by using XSL stylesheets. The prototype shown in this thesis focuses on mesh generators that produce structured meshes within a 2D domain with straight boundary edges.

D.1.2 Goals and Objectives

The test plan is written to discover and fix unexpected behaviors of PMG that may impact the functional requirements of PMG as listed in Appendix B. In the future, testers are expected to go through the test cases with appropriate techniques described in this document to uncover coding bugs and logic errors that affect the functional requirements, and possibly omissions in the implementation.

D.1.3 State of Scope

Because PMG is currently a prototype, we will focus on testing the functional requirements as listed in the SRS. The testing of nonfunctional requirements and system constraints, such as performance, usability, portability, etc is outside the scope of the test plan. The future improvements in these areas have been described as future work in Chapter 5.

D.2 Building the Test Plan

The rest of the test plan document is organized as follows. Section D.2.1 identifies the risks/concerns that need to be evaluated to assure they can be addressed by one or more testing techniques we will adopt. Section D.2.2 lists the components to be tested. Section D.2.3 discusses the testing strategy that we will adopt in this document, and provides the test matrix to show how each functional requirements can be tested with the strategies. Section D.2.4 describes the test cases in greater detail, where each test case will be described by the expected output versus corresponding inputs. When testing mesh generation, an exhaustive test is infeasible because of infinite possibilities. Therefore, we will run limited examples in that section. Finally Section D.2.5 provides an example of implemented test case that checks the output files contains the correct node order numbering for the element connectivity information.

D.2.1 Test Factors and Risks

During the development of a test plan, it is clear that the risk factors are the basis or objective of testing. In [1], the risks associated with testing are called test factors. Test factors are attributes of the software that, if they are required but not present, pose a risk to the success of the software. Depending on the understanding of our system PMG and the description of the 15 test factors in [1], we have chosen the following test factors that are most critical to the expected functionality of our software; therefore they must be tested. Tests must be selected to improve our confidence that the risks have been mitigated.

[1] Correctness. This test factor states that the functional requirements have been defined and is conformed by the developed design and implementation. The functional testing should ensure that all the functional requirements are properly implemented, etc.

[2] File integrity. The output file customization is an important system feature. Given a mesh, we should be able to test whether all the required information has been stored properly.

[3] Continuity of processing. Although implementing a full test on all possible error handling is not in our scope, the current prototype should provide certain degree of robustness, where the error detection and handling is reported by the XML and XSL parser.

[4] Maintainable. The reason we developed mesh generators as a program family includes that the program family methodology encourages program reuse. We should be able to test whether the system is prepared to be maintainable.

After identifying the set of test factors, we need to recognize the possible risks and build the test factor/test phase matrix. The test factor/test phase matrix indicates the correspondence between the specific risk scenarios and the test factor in different test phases. A risk scenario describes a specific unexpected behavior of the system. The following list discusses some potential risk scenarios that causes critical failures of our software.

[1] PMG generates an incorrect mesh generator according to a correct XML specification.

[2] For an MG produced by PMG, the values bound at specification time in the XML specification are changeable at run time.

[3] At specification time or run time, the specification of the value for the topology pattern does not correspond to the produced mesh.

[4] The output file produced by an MG contains incorrect information according to the corresponding XSL stylesheet. For example, the output file contains element connectivity when the XSL stylesheet instructs to only print vertex information, etc.

[5] PMG cannot detect and handle potential errors during the MG generation and mesh generation. For example, an unclosed boundary domain is meshed, although the corresponding MG should alert the user that the domain is incorrect and abandon the meshing process.

[6] Changing an individual module requires changes in many other modules of the system. For example, changing the meshing algorithm implementation to accommodate more topology pattern results changing the mesh drawing module, although using the same abstract representation can reduce the coupling between the two modules.

Table D.1 presents the test factor/test phase matrix given the test factors and the possible risk scenarios described earlier. For each risk scenario, its numeric identifier is cross-marked if the risk scenario can be checked against a test factor in a test phase. The possible test cases for PMG include system requirements, system architecture, implementation, unit testing, integration testing, and validation testing.

TFTP	SRS	MG	Impl	UT	IT	VT
Correctness	1,2,3		1,2,3		1,2,3	1,2,3
File Integrity	4		4		4	4
Continuity of Processing	5	5	5		5	5
Maintainable	6	6	6		6	6

Table D.1: Test Factor/Test Phase Matrix

D.2.2 Reference

[1] William E.Perry, Effective Methods for Software Testing. Second Edition. Wiley Computer Publishing, 2002

D.2.3 Tested Components

The following software components of PMG will be tested.

PMG

The PMG parsing needs to be tested to make sure the generated MG meets the requirement such that the specification-time bound values are read into the MG interface, and they are used to produce a correct mesh. At this step, An exhausted test is feasible by feeding possible XML specifications with different values of parameter of variation or the binding time. The resulting mesh generator should be able to contain the values specified in the XML specification, and have run-time values ready to be entered.

MG Interface Components

This test is to ensure the desired functionalities of the user interface is preserved. For each menu, the expected behavior can be visualized by the appearances of the resulting visual components. The following is a list of all the menu options that allow run-time inputs in the MG interface and their expected behaviors.

Interface Components to be Tested

- Initial Interface
- File menu and specification dialog

- Domain menu and specification dialog
- Geometry menu and specification dialog
- Element menu and specification dialog
- Boundary condition menu and specification dialog
- System parameter menu and specification dialog
- Output file menu and specification dialog
- Generate menu and mesh visualization frame

Meshing Algorithm and File Customization Components

The meshing algorithm component needs to be tested to ensure the implementation produces the correct mesh. One way to do this is through code inspection. Another way is that for each meshing problem that is tested, we feed the XML file containing the mesh data to another program that defines the properties that the mesh must have. Later in this document, we will show an example of testing node numbering with this approach.

Components to be Tested

- Meshing Algorithm
- XML file Writing
- XSL parser

D.2.4 Testing Methods

In this section, we will describe the methods we will use to perform the test. We will use four different testing methods to test our system.

D.2.4.1 Code Inspection

Code inspection is usually performed by the software developers as a formal analysis of the program source code to find defects. Small bugs can be detected by code inspection and they can be easily fixed. Code inspection also helps improves the readability and understandability of the program. The most critical programs in PMG are the XML specification parsing program and the mesh generation algorithm implemention. During the process of code inspection, the following checklist shows some standards regarding how the program source code of PMG can be checked.

- Are all the variables declared and initialized properly before they are used?
- Are the scope of each variable considered before they are used? A variable outside of its scope cannot be used.
- Is an array initialized with a hard-coded constant? If yes, replace the constant with a global constant if applicable.
- Is it possible an array of a certain size may be too small for extreme cases?
- Are all the functions and method calls consistent based on their access control identifiers? For example, a method declared public within a private class cannot be used by another class.
- Are all the function's return type consistently implemented? For example, does a function declared with non-void return type end with a return statement with a variable in the same type?
- Is there any type casting in the code? If yes, is it a possible cause for data accuracy loss?

– Is there any redundancy code in the program? For example, is there any function or variable that are never used? If yes, remove it.

– Is there any print statements irrelevant to system functions? If yes, remove it.

– Is there any non-constant variable whose value never changes? If yes, declare them as constants.

– Does a negative value for a variable make no sense? If yes, declare them as signed.

– If exception handling is implemented, are all try blocks followed by a catch block?

D.2.4.2 Unit Testing

In the unit test case, we will be testing the separate modules of the software. If the unit testing for all modules are conducted successfully, it is more likely that the system would meet its functional requirements. We will use both black-box and white-box test strategies. To test functional requirements, we can test the components by passing data through it, and we will be monitoring output and use it to compared with expected result. However, we have mentioned that an exhaustive test for mesh generation is not possible. Therefore, it is a good idea to use white-box strategy to test the mesh generation code. For example, the mesh generation code shown in Figure 3.6 can be used for boundary condition, edge, path, and statement check. A boundary condition check can be done by feeding a set of values for the rows, columns, subdivisions, etc. and test whether the code treats valid values as exceptions. For instance, the number of subdivisions number must be non-negative, otherwise, it should be an exception.

D.2.4.3 Integration Testing

In integration testing, we will pass data or control between components of the system. We will be looking for signs of the collision when multiple modules are involved to achieve a system function. Most test cases in Section D.2.4 will be done by integration testing. For example, the test of whether a value specified at specification value cannot be changed at run-time involves the XML parsing module and all the related interface module. The test of mesh correctness also requires the mesh generation algorithm and the file output module.

D.2.4.4 Validation Testing

In this method, we will work to find out if the software developed is valid according to the requirements. One way to perform a validation testing is to have several input data (XML specification) from which we will derive the results. We can compare the results from the PMG with the expected correct result to check the validation of the software. If there are problems with the system, we will record all the problems in a deficiency list. Eventually, all components and subcomponents of the system should be given a validation test.

D.2.4.5 Test Matrix

The test matrix is the key component of the test plan. First of all, it lists what is to be tested and which tests are to be performed. Secondly, between the two dimensions of the matrix are the tests applicable to the software. In our case, we are checking to ensure that requirements can be tested with one or more tests that we have mentioned. If a requirement can be checked with a test, it will be marked "T", otherwise it is blank.

The test matrix in Table D.2 only shows how the set of functional requirements as in Appendix B can be tested. The nonfunctional requirements and the system constraints are temporarily left out because they are outside the scope of the PMG prototype as explained in the body of the thesis.

Requirement	Code Inspection	UT	IT	VT
R1	T	T	T	T
R2	T		T	T
R3	T	T	T	T
R4			T	T
R5	T	T	T	T
R6	T	T	T	T
R7		T	T	T
R8		T	T	T
R9		T	T	T
R10	T	T	T	T
R11	T	T	T	T
R12	T	T	T	T
R13	T	T	T	T
R14	T	T	T	T
R15		T	T	T
R16	T	T	T	T
R17	T	T	T	T
R18	T	T	T	T
R19	T	T	T	T
R20	T	T	T	T
R21		T	T	T
R22		T	T	T

R23		T	T	T
R24	T	T	T	T
R25	T	T	T	T
R26	T	T	T	T

Table D.2: Test Matrix

D.2.5 Testing Cases

This section describes the detailed test cases. The test cases are aimed in breadth and will be divided into three groups depending on their relevance. The groups are PMG, MG interface, and mesh generation with output file. Each test case scenario is described with the following information: case identifier, test name, test type (UT, IT, or VT), inputs, expected behavior, expected output, and the associated requirement as in Appendix B to make sure all functional requirements are covered by test cases.

D.2.5.1 PMG

Test Case TC1

Name: XML parsing

Type: UT

Input: an XML specification according to the DTD. A fragment of the DTD is given in Figure 3.1.

Expected Behavior: XML parser customizes the object modeling the MG.

Expected Output: All the values specified in the XML specification can be output. The values for other variabilities are not assigned.

Associated Req: R1

Test Case TC2

Name: MG generation

Type: IT, VT

Input: an XML specification. The specification should contain at least the root element as defined by the DTD. The XML specification may contain any value that represents a parameter of variation as shown in Appendix A.

Expected Behavior: Parse the specification and produce an MG interface. Examples are shown in Chapter 4.

Expected Output: An MG interface with specification-time value fixed and run-time inputs waiting to be entered. If the specification only contains the root elements, all variabilities will be specified at run-time. Two examples are shown in Chapter 4.

Associated Req: R1,R2,R3,R4

Test Case TC3

Name: Geometry at spec time

Type: IT, VT

Input: an XML specification with a closed geometry specified at specification time. The geometry information is given in parametric form, which includes numbers of rows, columns, and vertex coordinates. An example spec is given in Figure 4.18.

Expected Behavior: XML parser recognizes the closed geometry value and customizes the object modeling the MG.

Expected Output: An MG interface with geometry values fixed, and other variabilities left blank. An example is shown in Figure 4.20.

Associated Req: R6,R7,R8,R22

Test Case TC4

Name: Domain Info at spec time

Type: IT, VT

Input: an XML specification with domain information specified at specification time. The domain info should include four strings: author name, date, application domain, and comment.

Expected Behavior: XML parser recognizes the domain information and customizes the object modeling the MG.

Expected Output: An MG interface with domain information fixed, and other variabilities left blank.

Associated Req: R5

Test Case TC5

Name: Element at spec time

Type: IT, VT

Input: an XML specification with element specification specified at specification time. The element specification includes the number of subdivisions, topology patterns, and node locations for geometry and dof.

Expected Behavior: XML parser recognizes the element value and customizes the object modeling the MG.

Expected Output: An MG interface with element values fixed, and other variabilities left blank.

Associated Req: R9,R11,R17,R18,R19,R20,R21

Test Case TC6

Name: Physical Attributes at spec time

Type: IT, VT

Input: an XML specification with physical specification specified at specification time. The physical specification includes boundary conditions and material properties described by name,location,and value.

Expected Behavior: XML parser recognizes the physical attributes specification and customizes the object modeling the MG.

Expected Output: An MG interface with element values fixed, and other variabilities left blank.

Associated Req: R10,R12,R13,R14,R15

Test Case TC7

Name: System Parameter at spec time

Type: IT, VT

Input: an XML specification with system parameter specification specified at specification time. A system parameter is described by a name and the value.

Expected Behavior: XML parser recognizes the system parameter information and customizes the object modeling the MG.

Expected Output: An MG interface with system parameter information fixed, and other variabilities left blank.

Associated Req: R16

Test Case TC8

Name: Output File at spec time

Type: IT, VT

Input: an XML specification with output file specification specified at specification time. The locations of the XSL stylesheet must be given.

Expected Behavior: XML parser recognizes the output file information and customizes the object modeling the MG.

Expected Output: An MG interface with output file information fixed, and other variabilities left blank.

Associated Req: R23,R24,R25,R26

Test Case TC9

Name: Inputs at Run time

Type: IT, VT

Input: an XML specification with only root element

Expected Behavior: XML parser should produce the most-general purpose mesh generator

Expected Output: An MG interface where all variabilities are left blank, so their values are entered at run-time. Figure 4.3-4.9 shows an example.

D.2.5.2 MG Interface

This section discusses the test procedures to verify the appearances of the MG interface. The purpose of testing the interface is to ensure that all values that may be specified at run-time is properly represented and can be entered in the interface.

Test Case TC10

Name: File Dialog

Type: IT,VT

Input: an XML specification

Expected Behavior: The MG interface should allow for basic file open/save functionality.

Expected Output: "File" menu options provide "open" and "save".

Associated Req: –

Test Case TC11

Name: Domain Dialog

Type: IT,VT

Input: an XML specification

Expected Behavior: The MG interface should allow the entering of domain information.

Expected Output: The domain dialog has labels and boxes so user can enter the domain information.

Associated Req: –

Test Case TC12

Name: Geometry Dialog

Type: IT,VT

Input: an XML specification

Expected Behavior: The MG interface should allow the entering of geometry information.

Expected Output: The geometry dialog has labels and boxes so user can enter the domain information.

Associated Req: –

Test Case TC13

Name: Element Dialog

Type: IT,VT

Input: an XML specification

Expected Behavior: The MG interface should allow the entering of element information.

Expected Output: The element dialog has labels and boxes so user can enter the element information.

Associated Req: –

Test Case TC14

Name: physical attributes Dialog

Type: IT,VT

Input: an XML specification

Expected Behavior: The MG interface should allow the entering of physical information.

Expected Output: The boundary condition and material dialog have labels and boxes so user can enter the relevant information.

Associated Req: –

Test Case TC15

Name: System Parameter and File Dialog

Type: IT,VT

Input: an XML specification

Expected Behavior: The MG interface should allow the entering of system parameters and file specification.

Expected Output: The system parameter and file dialog have labels and boxes so user can enter the relevant information.

Associated Req: –

Test Case TC16

Name: Information Passing

Type: IT,VT

Input: XML specification

Expected Behavior: The MG interface should provide functionalities so that the run-time inputs can be passed in to produce the mesh.

Expected Output: The "OK" and "Cancel" button are typically used.

Associated Req: –

D.2.5.3 Mesh Generation and Output File

Test Case TC17

Name: Produce Mesh

Type: IT,VT

Input: an XML specification with the configuration as Figure 4.2.2 for quadrilateral mesh

Expected Behavior: Produce a quadrilateral Mesh

Expected Output: A mesh is produced and can be visualized. The mesh is shown in Figure 4.10.

Associated Req: R25,R26

Test Case TC18

Name: Different Topology

Type: IT,VT

Input: The same XML specification as above, but with QUAD8 pattern at specification time or run time

Expected Behavior: Produce a Mesh within the same domain in the QUAD8 pattern

Expected Output: A mesh is produced and can be visualized. The mesh is shown in Figure 4.11.

Associated Req: R26

Test Case TC19

Name: Check Topology

Type: IT,VT

Input: an XML specification

Expected Behavior: All elements reflect the same topology

Expected Output: True if all the element reflect the required topology

Associated Req: R26

Test Case TC20

Name: Check XML file

Type: IT,VT

Input: An XML specification

Expected Behavior: Use external programs to check the correctness of mesh data. Section 2.4.4 has an example of checking the counterclockwise of all elements.

Expected Output: True when the test is successful, and false otherwise

Associated Req: R23

Test Case TC21

Name: Produce Output File

Type: IT,VT

Input: an XML specification the same as the previous test case, and the stylesheets as shown in Figure 4.12 and 4.13.

Expected Behavior: Produce two text files with vertex and connectivity information.

Expected Output: The files as shown in Figure 4.15 and 4.16.

Associated Req: R23

Test Case TC22

Name: Different Output File

Type: IT,VT

Input: an XML specification the same as the previous test case but with QUAD8 pattern, and the stylesheets as shown in Figure 4.12 and 4.13.

Expected Behavior: Produce two text files with vertex and connectivity information.

Expected Output: The first file should have contain new vertex information, and the second file should have four times the number of elements as the previous test case.

Associated Req: R23

Test Case TC23

Name: Different Material Property

Type: IT,VT

Input: an XML specification

Expected Behavior: Changing the value of a material properties results changing the corresponding contents of the output file

Expected Output: By inspecting the file contents, one can tell whether changes made at either specification time or run time have been reflected in the new output file.

Associated Req: R23

D.2.6 Detailed Test Cases

This section lists six detailed test cases. For each of the six test cases, we give all of the information necessary to do the test. Each test case is explained step by step, followed by the expected behavior and outputs. The last test case in this section explains an automated testing, which may require a separate engine.

Test Case TC24

Name Test General Purpose Mesh generator

Purpose Check that all variabilities should be specified at run-time.

Input An XML specification with only the root elements. According to the DTD, the XML file with the following contents should be prepared first. The test is done by assuming the file name is meshml.xml and it has been saved in the program directory.

Test Procedure

```
<?xml version="1.0"?>
<MeshGeneratorApplication>
</MeshGeneratorApplication>
```

Figure D.1: XML Specification for General-purpose Mesh Generator

[1] In the DOS window, go to the correct directory the program is in.

[2] Following the prompt, type in "java jaxp file:/thedirectory/meshml.xml

[3] The most general-purpose mesh generator interface will appear. It is the same as shown in Figure 4.3.

[4] Within the interface, click "domain" menu. The domain dialog will show up. The dialog boxes should all be empty and accessible. Click "OK" or "Cancel" to close dialog.

[5] Click "geometry" menu, the geometry dialog will show up. The boxes to specify the number of rows and column, the zone id and material, and the vertex ID and coordinates should be empty and accessible. Click "OK" or "Cancel" to close dialog.

[6] Click "element" menu, the element dialog will show up. Under the first tab "element", the number of subdivisions along each direction should be empty and accessible. The topology pattern should display "Quadrilateral". Under the second tab "node", all the node specification boxes should be empty and accessible. They include the number of nodes for the geometry and the dof, and their locations. The names of dofs can also be specified. Click "OK" or "Cancel" to close dialog.

[7] Click "boundarycondition" menu, the boundary condition dialog will show up. The dialog should be with a vertical bar, and for each boundary condition, its name, location, and value should be empty and accessible. Click "OK" or "Cancel" to close dialog.

[8] Click "material" menu, the material dialog will show up. Click on each of the five tabs, the names and values of each material property is empty and editable. The value in the type box should all be "integer". Click "OK" or "Cancel" to close dialog.

[9] Click "system parameter" menu, the system parameter dialog will show up. For each system parameter, its name and value should be empty and editable. Click "OK" or "Cancel" to close dialog.

[10] Click "output file" menu, the output file dialog will show up. For each output file, its location box should be empty and editable. Click on "browse" button to select a file by mouse. Click "OK" or "Cancel" to close dialog.

Test Case TC25

Name Test Special Purpose Mesh generator

Purpose Check that the variabilities specified at specification time cannot be changed at run-time, and other variabilities are left blank.

Input An XML specification with material property and boundary condition fixed. According to the DTD, the XML file with the following contents should be prepared first. The test is done by assuming the file name is meshml.xml and it has been saved in the program directory.

Test Procedure

```
<?xml version="1.0"?>
<MeshGeneratorApplication>
<MaterialSet>
    <material>
        <matproperty>
            <name>Poisson's ratio</name>
            <type>real</name>
            <value>0.3</value>
        <matproperty>
    <material>
</MaterialSet>
<loadcase>
    <boundarycondition>
    <name> traction </name>
    <location> 0,1  </location>
    <value>   3.0E6 </value>
    </boundarycondition>
</loadcase>
</MeshGeneratorApplication>
```

Figure D.2: XML Specification for Special-purpose Mesh Generator

[1] In the DOS window, go to the correct directory the program is in.

[2] Following the prompt, type in "java jaxp file:/thedirectory/meshml.xml

[3] The mesh generator interface will appear. It is the same as shown in Figure 4.3.

[4] Within the interface, click "domain" menu. The domain dialog will show up. The

dialog boxes should all be empty and accessible. Click “OK” or “Cancel” to close dialog.

[5] Click “geometry” menu, the geometry dialog will show up. The boxes to specify the number of rows and column, the zone id and material, and the vertex ID and coordinates should be empty and accessible. Click “OK” or “Cancel” to close dialog.

[6] Click “element” menu, the element dialog will show up. Under the first tab “element”, the number of subdivisions along each direction should be empty and accessible. The topology pattern should display “Quadrilateral”. Under the second tab “node”, all the node specification boxes should be empty and accessible. They include the number of nodes for the geometry and the dof, and their locations. The names of dofs can also be specified. Click “OK” or “Cancel” to close dialog.

[7] Click “boundarycondition” menu, the boundary condition dialog will show up. The dialog should have one boundary condition fixed as gray. The corresponding name, location, and value are “traction”, “0,1”, and “3.0E6”. The values are not editable. Click “OK” or “Cancel” to close dialog.

[8] Click “material” menu, the material dialog will show up. there should be one tab with one material property available. The name, type, and value for the material property are “Poisson’s ratio”, “Real”, and “0.3”. They are not editable. Click “OK” or “Cancel” to close dialog.

[9] Click “system parameter” menu, the system parameter dialog will show up. For each system parameter, its name and value should be empty and editable. Click “OK” or “Cancel” to close dialog.

[10] Click “output file” menu, the output file dialog will show up. For each output file, its

location box should be empty and editable. Click on "browse" button to select a file by mouse. Click "OK" or "Cancel" to close dialog.

Test Case TC26

Name Generate a mesh with different topologies

Purpose Check that changing topologies at run time changes the mesh

Input We will use the same specification from the previous test case. All other variabilities are set at run time. The domain is a unit square with 4 zones.

Test Procedure

[1] In the DOS window, go to the correct directory the program is in.

[2] Following the prompt, type in "java jaxp file:/thedirectory/meshml.xml"

[3] The mesh generator interface will appear. It is the same as shown in Figure 4.3.

[4] Click "geometry" menu, the geometry dialog will show up. We specify the following information:

- number of rows is 2;
- number of columns is 2;
- The first zone id is 1. the vertices are specified as (0,0), (1,0), (1,1), (0,1)
- The first zone id is 2. the vertices are specified as (0,0), (1,1), (1,1), (0,1)
- The first zone id is 3. the vertices are specified as (1,0), (2,0), (2,1), (1,1)
- The first zone id is 4. the vertices are specified as (1,1), (2,1), (2,2), (1,2)

[5] Click "element" menu, the element dialog will show up. Under the first tab "element", we specify the following:

- number of subdivisions in r direction is 2,2
- number of subdivisions in s direction is 5,3
- topology pattern is Triangle1

Under tab "node", we need to specify the following:

- number of nodes for geometry and dof are both 3
- node 1,2,3 are at location (1,0,0),(0,1,0),(0,0,1). The name of dof is "x displacement".

[6] Click "boundarycondition" menu, the boundary condition dialog will show up. The dialog should have one boundary condition fixed as gray. The corresponding name, location, and value are "traction", "0,1", and "3.0E6". The values are not editable. Click "OK" or "Cancel" to close dialog.

[7] Click "material" menu, the material dialog will show up. There should be one tab with one material property available. The name, type, and value for the material property are "Poisson's ratio", "Real", and "0.3". They are not editable. Click "OK" or "Cancel" to close dialog.

[8] Click "Generate" and a new frame with the mesh is displayed. The pattern can be seen as Triangle1 because the triangle are divided by going right. The mesh is also dense vertically because the number of subdivisions are greater than horizontally.

[9] Go back to the interface and go to the element dialog, change the pattern to Triangle8 and click "Generate". The resulting mesh should be of the corresponding pattern by examining all the elements.

Test Case TC27

Name File Correctness

Purpose Check the content of the output file

Input This test case is an extension of the previous test case. We can start with the same specification and go through all the steps as described. This is the starting point of this test case. The other inputs used in this test are the stylesheet file given partially as follows. The stylesheet instructs to print the material and boundary condition information.

Test Procedure

```
<xsl:template match="Mesh">
<xsl:text>ID&#x09;x_coord&#x09;y_coord&#x09;</xsl:text>
<xsl:value-of select="$newline"/>
<xsl:apply-templates select = "MaterialSet"/>
</xsl:template>
<xsl:template match="MaterialSet">
<xsl:for-each select = "material">
<xsl:value-of select = "name"/> <xsl:value-of select = "$tab"/>
<xsl:value-of select = "value"/> <xsl:value-of select = "$tab"/>
<xsl:value-of select = "$newline"/>
</xsl:for-each>
</xsl:template>
<xsl:apply-templates select = "boundaryconditionset"/>
</xsl:template>
<xsl:template match="boundaryconditionset">
<xsl:for-each select = "boundarycondition">
<xsl:value-of select = "location"/> <xsl:value-of select = "$tab"/>
<xsl:value-of select = "name"/> <xsl:value-of select = "$tab"/>
<xsl:value-of select = "value"/>
<xsl:value-of select = "$newline"/>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Figure D.3: Stylesheet File

[1] Go to th element dialog and under the “element” tab, choose the pattern QUAD. This will generate a quadrilateral mesh.

[2] Go to the output file dialog and specify the location of the stylesheet. This can be done either by typing manually or use the browse button.

[3] Click "Generate" to generate a quadrilateral mesh.

[4] The generated output can be opened to check contents. First of all, the material information should be printed as "Poisson's ratio", "0.3". The boundary conditions contains the vertices for the traction as specified. In this file, the traction should be specified for vertex ID 1, 10, 19 since these three vertices are on the edge 0,1 as specified. No other vertex should be printed in this file.

Test Case TC28

Name File Correctness for minimum bandwidth

Purpose Check the content of the output file

Input This test case is another one extended from previous case. This test case is an simple example of white box testing and check whether the vertex numbering is horizontally or vertically depending on the number of subdivisions. Our program implemented this feature for all topology patterns except for Triangle8. For the simplicity of file checking, the number of subdivisions horizontally will be 2,2 and 2,3 vertically. The XSL file used in this case lists the vertex ordering, the same as in Figure 4.12. This test case is an coverage testing derived from the code such that if the number of subdivisions on one direction is greater than the other, the vertex numbering in the output file will be in the corresponding direction.

Test Procedure

[1] Start from the previous interface and keep the topology pattern as QUAD.

[2] Change the number of subdivisions to 2,2 in s axis and 2,3 for r axis. The mesh

generation algorithm will choose the path leading to vertices numbering horizontally first, followed by vertically.

[3] Click "Generate" to generate the quadrilateral mesh. The mesh should contain fewer elements than the previous case.

[4] Check the output file that contains the vertex coordinates. Every group of six vertices should be having the same y coordinate with increment on the x coordinate. This is because the number of subdivisions for vertical direction is greater than the horizontal direction.

[5] To test the opposite case, go back to element dialog and switch the input values for the subdivision numbers of the two directions, and generate the mesh. The output file should list the vertices in the way that every group of six vertices have the same x coordinate with increment on the y coordinate.

Test Case TC29

Name Automated Testing

Purpose Check the mesh output files based on the XML specification

Input This test case requires an XML specification and an input of an simulation engine we can call "PMGtester" from a slight modification of the program. The MG interface will not be visible. PMGtester will have a component called "PMGFileChecker" to check the contents of the output files. In terms of the specification, the following inputs are specified in the specification.

– Geometry

[1] number of rows is 2, number of columns is 2

[2] For each of the four zones, specify the vertex coordinates as in TC26.

– Element

[1] number of subdivisions is 2,2 for each direction, and the pattern is "Triangle1"

[2] Specify the node information as in TC26.

– Physical attributes. We can reuse the boundary condition specification as in TC25.

– Output files. We specify none. We will check the XML data file in this case.

Test Procedure

[1] Develop PMGtester.

[2] Name the specification "meshml.xml". Under the DOS window, start PMGtester and use it to read meshml.xml

[3] The mesh data file will be named "MeshData.xml". In this case, PMGFileChecker will read MeshData.xml to check the following information.

- Each element should have three vertices.
- The counterclockwise property can be checked as in Section 2.7.
- The vertex coordinates cannot be negative.
- The material property should be included that has "Poisson's ratio", "real", and "0.3".
- The number of vertices should be 25.
- The number of elements should be 32.

  - The boundary conditions should be specified for vertex 1,6,11.(The row numbering is used by default)

[4] We can automate this process by changing the topology pattern to "Triangle1" to "Triangle8" in the specification. PMGFileChecker will be used to test different output file to verify the contents. For example, if "triangle8" is used, the number of vertices should be 41 given the same subdivisions. However, changing the pattern to "QUAD" requires changing the node information. Each element should be given 4 nodes instead of 3. The resulting output file should have 25 vertices and 16 elements.

[5] PMGFileChecker is also capable to check correct vertx numbering if we change the subdivision to unequal numbers.

D.2.7 Example Test Case

This section illustrates an example to test the validity of the mesh by examining whether the node numbering of each element is counterclockwise. We chose to implement this test case because this is a nontrivial test given that correctness is the focus of our testing. The correct order of vertices in a mesh is a very important property that each mesh in our program family must hold true. A mesh that violates this property means that our software is incorrect because it may cause invalid results from the finite element program.

This test case is implemented as follows: We will write a separate program that examines the XML file that contains the mesh data. The connectivity of each element will be checked by using the basic computational geometry theory. Given the time constraints, we will only implement the triangular elements in this document. The case of quadrilateral elements can be done as part of the future work.

The test of whether the vertices of a triangular element are counterclockwise is given by the following piece of code as part of a simple XML parser. The array variable "set" contains the three vertices for each element. The vertices within the array are stored in the order as in the XML data file indicated by the element connectivity information. The theory of the computation is adapted from Triangle software implementation. (URL available at http://www.cs.cmu.edu/ quake/triangle.html) We used this parser to test the following mesh

```
public double counterclockwise(Vertex[] set)
{
double result, resultleft, resultright = 0.0;
resultleft  = (set[0].x_coordinate - set[2].x_coordinate)*
              (set[1].y_coordinate - set[2].y_coordinate);
resultright = (set[0].y_coordinate - set[2].y_coordinate)*
              (set[1].x_coordinate - set[2].x_coordinate);
result = resultleft - resultright;
return result;
}
```

Figure D.4: Test Code for Counterclockwise

data XML files. The picture of the mesh and the test result are also given.

The test runs through the connectivity information in the XML data file and use the counterclockwise function. The result is shown as passed so this is a correct data file. We also run the same test for another data file corresponding to the irregular domain mesh as shown in Chapter 4, and we get the same results.

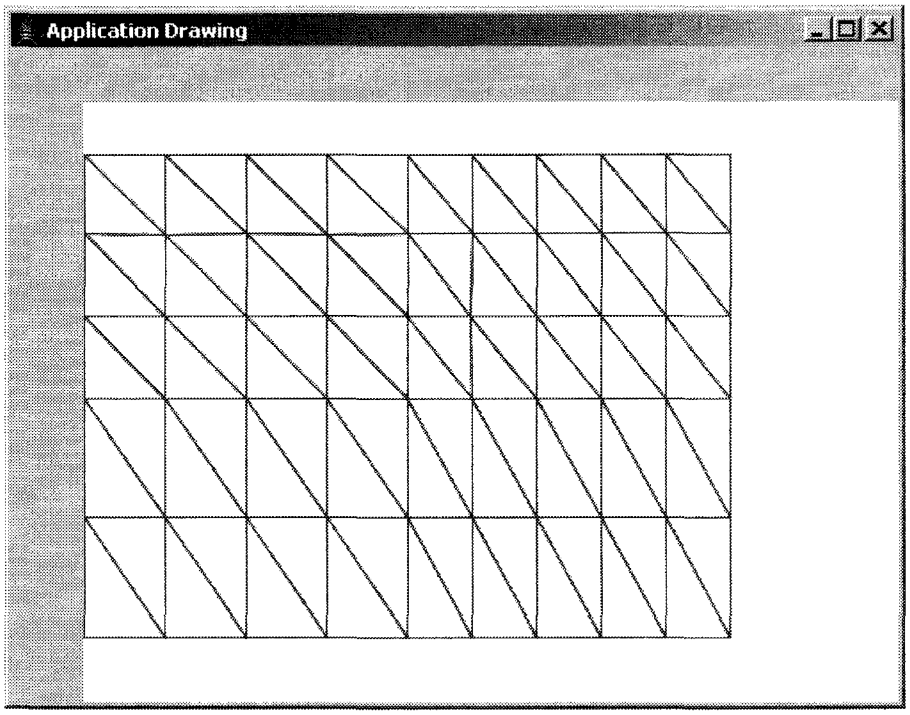
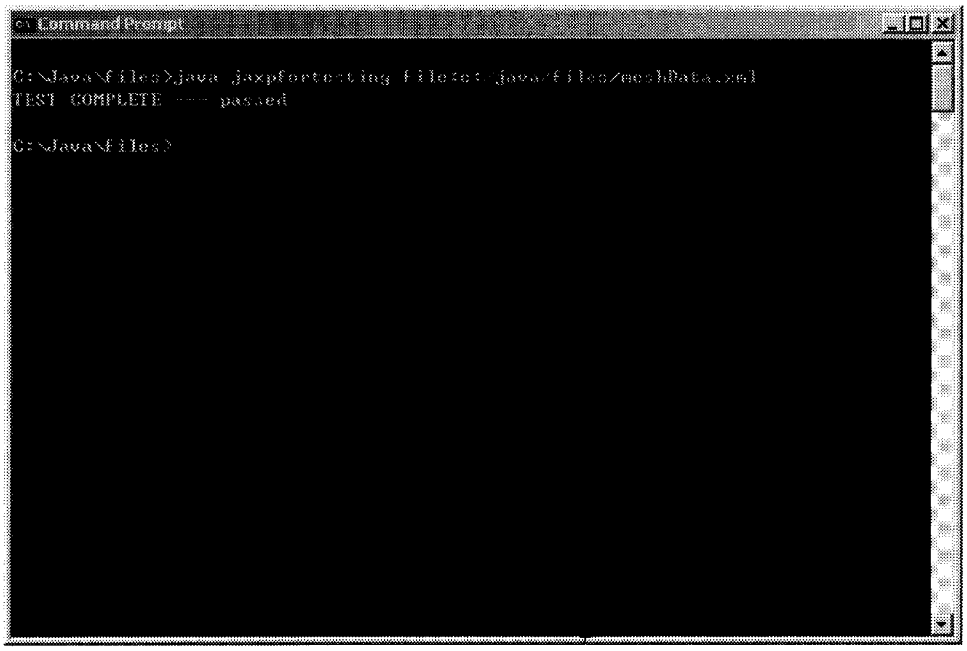


Figure D.5: Triangular Mesh Used for Testing



```
C:\Java\files>java jaxpfortesting file:c:/java/files/meshData.xml
TEST COMPLETE --- passed

C:\Java\files>
```

Figure D.6: Test Result for the Mesh

5606 61