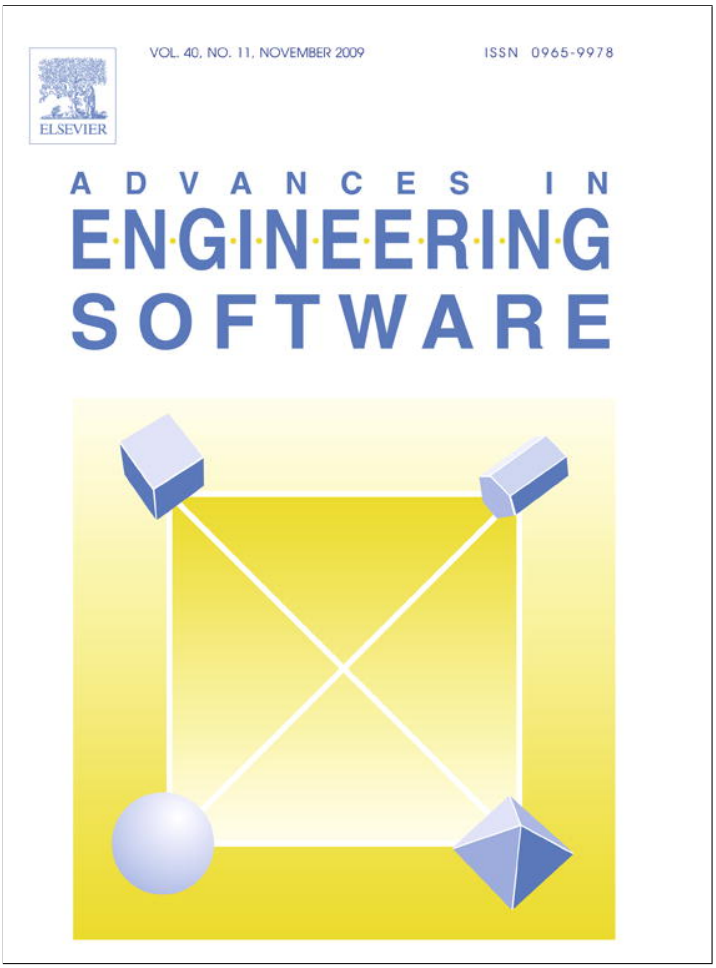

Advances in Engineering Software 40 (2009) 1155–1167

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

A document driven methodology for developing a high quality Parallel Mesh Generation Toolbox

S. Smith *, W. Yu

Computing and Software Department, McMaster University, Hamilton, Ontario, Canada L8S 4L7

ARTICLE INFO

Article history:
Received 9 November 2007
Received in revised form 28 January 2008
Accepted 11 May 2009
Available online 26 June 2009

Keywords:
Mesh generation
Software engineering
Software quality
Scientific computing

ABSTRACT

This paper motivates the value of using a document driven methodology to improve the quality of scientific computing applications by illustrating the design and documentation of a Parallel Mesh Generation Toolbox (PMGT). Formal mathematical specification is promoted for writing unambiguous requirements, which can be used to judge the correctness and thus the reliability of PMGT. Mathematics is also shown to improve understandability, reusability and maintainability through modelling software modules as finite state machines. The proposed methodology includes explicit traceability between requirements, design, implementation and test cases. Traceability improves the verification of completeness and consistency and it allows for proper change management. To improve the reliability of PMGT, given the challenge that the correct solution is unknown a priori, an automated testing approach is adopted to verify the known properties of a correct solution, such as conformality and counterclockwise vertex numbering.

1. Introduction

The field of scientific computing (SC) has developed an impressive variety of algorithms and libraries that have greatly influenced and improved the work of scientists and engineers. SC software has, for instance, successfully been used to increase the productivity of manufacturing processes, to improve the effectiveness of health care treatments and to raise the level of safety obtained by new building and vehicle designs. However, instances have occurred where the confidence placed in SC software has been misplaced. For instance, Oliveira and Stewart [21] summarize three examples of SC software failures: (i) the 1991 failure of a Patriot missile to hit an incoming Scud missile; (ii) the 1996 explosion of the Ariane 5 rocket; and, (iii) the 1991 collapse of the Sleipner A oil rig. All of these examples led to significant financial losses, and in the case of the Patriot missile disaster, to a tragic loss of human life. The disturbing possibility of relying on incorrect SC software is further highlighted by the errors found when comparing nine large commercial packages for seismic data processing [14,15]. All of the packages were initially based on the same mathematical algorithms, but in experiments they produced different results, with an average absolute difference of 1% per 4000 lines of code [15]. The above examples illustrate that problems exist with the software quality of correctness, even though correctness is generally considered to be the most important quality in SC [19]. If this most important quality is not being achieved, then it is reasonable to believe that other software qualities, such as understandability, maintainability and reusability, are also suffering.

How can the quality of SC software be improved? The idea promoted in this paper is to adapt methodologies that have been successfully used in the field of software engineering (SE). SE methodologies are now commonly employed in developing business applications, information systems and real-time safety critical systems, such as the shutdown system for the Darlington nuclear generating station [29] and the US Navy's A-7E aircraft [16].

SC application developers often have a science or an engineering background, with its associated emphasis on reproducibility, documentation, rigour and the scientific method; therefore, a systematic SE approach should appeal to them. However, SE methodologies, in particular software development tools, are not commonly employed in SC [48]. Moreover, as Segal observes "there is no discrete phase of requirements gathering or of software evaluation. Testing is of the cursory nature which would enrage a software engineer" [35]. Segal's description of the typical SC development process fits with a recent survey of approximately 160 SC software developers [43], which found that requirements specification only occupies about 12% of the total development time and that 90% of developers do not use formal specifications, which, according to some prominent SE researchers, are crucial for developing high quality software.

Use of SE methodologies in SC is not as simple as just opening a textbook and adopting existing methods and techniques. As Segal's field studies [35] show, interaction between SE and SC can be problematic because each side fails to meet the expectations of the

* Corresponding author. Tel.: +1 905 525 9140x27929; fax: +1 905 524 0340.
E-mail address: smiths@mcmaster.ca (S. Smith).

0965-9978/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2009.05.003

other side. This gap between SE and SC has been termed a "software chasm" [18]. The chasm exists in part because most SE methods and techniques do not directly map to SC applications, since the characteristics of SC software differ from those of the business and real-time systems that SE research has tended to focus on [39]. A simple example of this can be found in the common SE advice to use pass by value when a variable will not be modified by a subroutine. Although this is a strategy that SE promotes to keep data safe from change, if the large arrays used in SC are passed by value, then an unnecessarily large block of memory on the stack will be consumed [21, page 51]. Performance considerations will require that most SC software ignore SE advice and adopt pass by reference for arrays.

The disconnect between SE research and SC applications is widened by the fact that there are so few examples available in the literature that illustrate how SE methodologies can be adapted to SC. This paper attempts to address this problem by proposing new SE inspired documents tailored for SC software development. The proposed documentation requires an up-front investment of time and effort, but this effort should improve the quality of both the development process and the final product. This statement is supported in the sections that follow through arguments that appeal to engineering judgement and to the success of similar techniques for other types of software.

The example used to illustrate the proposed methodology and associated sequence of documents is a Parallel Mesh Generation Toolbox (PMGT). PMGT manages the geometry and topology information for a mesh that is potentially stored across multiple processors. This application was chosen as a case study in the application of SE methodologies because high quality software to manage mesh data is of value in many applications. In particular, a quality mesh generator is often necessary for the numerical solution of partial differential equations (PDEs), especially when mesh adaptivity is used because the mesh data will require frequent modification. The full design and documentation of PMGT is presented in [49]. The focus of this work is not on developing an improved mesh generation algorithm, but rather on providing an example of how to design and document complex SC software.

The first section below lists the software qualities of interest for PMGT, provides an explanation for their importance to SC, and describes the challenges one faces in achieving them. The next section outlines the proposed documentation. The sections that follow describe the documentation for requirements, high level design, low level design, code and test cases.

2. Software qualities

In the current work, quality is defined as a measure, potentially phrased in relative terms, of the excellence or worth of a software product or process. Although quality is often defined as customer satisfaction or as meeting requirements, these definitions do not capture the fact that customers can be satisfied and have their requirements met by a low quality product. For instance, in some SC applications a low accuracy may be tolerated. Although higher accuracy represents higher quality, the additional expense, or the necessary sacrifice in performance (speed), may mean that the lower accuracy solution is preferable. This example highlights the fact that quality is not a single measure but rather a collection of measures. For a given software product one can require, verify and measure such qualities as accuracy, performance and reliability. These qualities and others of interest for the PMGT library and for many SC applications are defined as follows:

Reliability: The probability that the software will meet its stated requirements under a given usage profile. As an example, an SC application is reliable if the true error is rarely larger than the user requested bound on the error.
Accuracy: A measure of the absolute or relative error of the approximate solution compared to the true solution.
Performance: A measure of how large a problem can be handled and how quickly an answer can be found. Performance is related to how efficiently the internal resources of the computer are used.
Understandability: The degree to which a programmer finds the software library easy to understand.
Maintainability: The effort required to locate and fix errors and to add features to an operational program.
Verifiability: The effort necessary to verify the properties of a software system.
Reusability: The extent to which a program can be used in other applications.
Portability: The effort required to transfer the software from one operating environment to another.

The above definitions are adapted from [12,20], with the definitions for reliability and accuracy derived from [9]. To highlight the qualities of interest in the remainder of the paper, the convention will be adopted that the software qualities will be capitalized and displayed in an *italic* font.

The importance of writing reliable software is highlighted in this paper's introduction (Section 1) through examples of incorrect SC software and the incorrect usage of SC software. Reliability is not the only software quality where there is room for improvement in SC software. Reusability is another software quality that can be improved, especially since reuse of reliable components improves confidence in the computed solutions. Although there are significant success stories in SC with respect to reuse, with many popular libraries in existence, there are still some developers that insist on "reinventing the wheel." As Dubois [8] observes, component reuse is sometimes poor in SC, even when good mathematical libraries are available, because programmers often refuse to believe that the implementation needs to be as complicated as it is in the existing code. The difficulty of achieving reusability is evident by the large number of similar SC packages that have been developed. For instance, for mesh generation software, [23] identifies 94 software packages, 61 of which generate triangular meshes, with 43 of these using a Delaunay triangulation algorithm.

The discussion above focuses on problems with product qualities, but room also exists for improving the quality of the process of developing SC software. For instance, the process should be usable, reusable and provide high performance. In the context of the process, the quality of performance if often termed productivity and refers to the efficient use of human, computer and time resources toward solving SC problems. The importance of improving productivity in SC is underscored by the recent creation of the Defense Advanced Research Project Agency (DARPA) High Productivity Computing Systems (HPCS) program [42].

Developing high quality SC software is challenging. For instance, judging the *Accuracy* and thus the *Reliability* of such software is difficult due to the lack of an expected true solution. Unlike many other classes of software the true solution to SC problems often cannot be determined a priori. In fact, the purpose of the software is to solve problems that are difficult or impossible to solve any other way. Another challenge is that most real numbers cannot be represented exactly on a computer and have to be approximated by floating point numbers. Round off error makes it difficult for SC software to achieve the qualities of *Accuracy* and *Reliability*. Additional storage may be used for floating point numbers to improve their precision and thus the *Accuracy* of the computed solution. However, this decision will lower the *Performance* of the software. This example illustrates the chal-

lenge of accounting for the tradeoffs between competing qualities in SC.

To further illustrate the tradeoffs between different qualities, *Performance* can be improved by taking advantage of parallel computation, but using multiple processors can potentially lower other software qualities. For instance, *Reliability* may suffer because "most programming models for parallel computers have made a deliberate choice to present opportunities for performance at the cost of a greater chance for programmer error [9, page 258]." Moreover, the sheer volume of calculations in a large parallel computation makes the low probability of hardware error significant enough that it could affect *Reliability* [9, page 262]. In code for parallel computations, the qualities of *Understandability*, *Maintainability*, and *Portability*, can all potentially suffer because it is difficult to abstract away the implementation decisions from the program interfaces and from the algorithms. In addition, *Verifiability* is particularly difficult to achieve for parallel programs [36].

3. Software documentation

SE methodologies can potentially improve the quality of SC, but just using any SE methodology will not necessarily lead to improvement. Although SE has promoted domain independent methodologies, success in SC likely requires a domain specific methodology [18]. The inadequacy of the "one size fits all" approach is illustrated by a case study [35] that shows how a domain independent waterfall process led to clashes between software engineers and computational scientists. The linear waterfall model is difficult for SC because it is hard to initially know what assumptions are valid and what algorithms will have the desired convergence and stability properties.

Although the methodology described below is structured around documents that correspond to phases in the waterfall model, the form of the documents has been customized to SC. Moreover, the associated development process does not have to generate the documents in distinct sequential stages. The process of SC software development will not usually proceed as rationally as the ideal waterfall model, but the advantages of a rational process can still be obtained by documenting the work products as if they were developed and written following the waterfall model [28]. The advantage of documentation that follows the waterfall model is that it closely parallels how engineers typically think about their work flow. Moreover, the software lifecycle model and the scientific method can both be abstractly modelled using a waterfall process [40], which should make the proposed sequence of documents familiar to SC professionals.

In this paper the development of the software is called document driven, as the separate stages of the idealized waterfall, shown in Fig. 1, are identified by the documents produced at each stage, as follows: Commonality Analysis (CA), Software Requirements Specification (SRS), Module Guide (MG), Module Interface Specification (MIS), code, and Summary of Validation Testing Report (SVTR). The back and forth arrows represent the iterative process of validation and derivation. Before the project is complete, the code needs to be validated against the design and against the requirements, as represented by the dashed arrow from "SVTR" to "SRS." Although the validation cannot occur until after the implementation is complete, the plan for the SVTR document can be developed at the same time as the SRS, since succeeding in the software validation is always part of the requirements.

The development of SC software will likely be highly iterative, so the documentation must support change. For this reason explicit traceability is included between the stages of the waterfall; that is, the connections between the requirements, design, code and test cases are identified, documented and verified. The components of each of the different design documents are labelled and traceability matrices are built to document the relationship between the components. This traceability assists with the *Verifiability* of completeness and consistency. This in turn potentially improves *Reliability* because the traceability explicitly shows that the initial requirements for PMGT have been implemented by the identified modules and tested by the likely test cases. Traceability also facilitates proper change management because the connection between the design and anticipated changes is explicitly documented. Thus traceability improves *Maintainability*. Traceability is very important for an iterative process because all of the documents will change as the focus moves back and forth between them.

The first step in the software development process is a Commonality Analysis (CA), where a CA studies shared features or attributes common among similar software programs so that the software can be developed as a program family. The idea of program families was introduced by Dijkstra [7] and later investigated by Parnas [25,26]. More recently, Weiss [1,47] has considered the concept of a program family in the context of what he terms Family oriented Abstraction, Specification and Translation (FAST) [6,46]. Other approaches to developing program families,

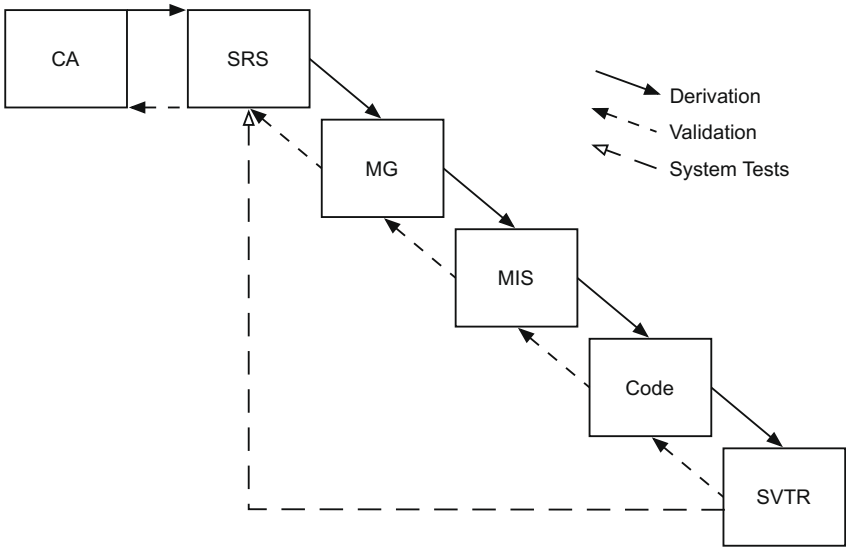

Fig. 1. Proposed documentation.

also known as software product lines, can be found in [5,30]. The use of a CA in the context of general purpose SC is discussed in [37]. A CA for a program family documents the terminology, commonalities (including goals and theoretical models) and variabilities (including assumptions, input variabilities and output variabilities). The CA for a program family of mesh generators is presented in [38]. The more general idea of development of mesh generators as a program family, or product line, is discussed in [2,3,41].

The CA for a mesh generator [38] can be refined, starting with the SRS, into various program family members and their associated documentation. For developing PMGT there were two family members. The first, the basic core product, was a sequential program for grid refinement. The second family member added parallelism. The development of the second program was made much easier because of the *Reusability* of the documentation from the first program. The development of the second program took one developer (Yu) about 9 months to complete (3 months for the SRS, 1 month for the MG, 2 months for the MIS, 1.5 months for the implementation and 1.5 months for the SVTR). The time for development includes time spent on repeated iterations of each of the documents, time for learning mesh generation algorithms and time spent learning how to document the requirements and the design. Not including code for testing, the serial version of PMGT is about 3000 lines of code and the parallel version adds about 1000 lines to this. By way of comparison, the Basic COCOMO [4, page 75] estimate for a 4000 line (KLOC = 4) semi-detached program is 14 months of development time E, where $E = a_b \text{KLOC}^{b_b} = 3.0 \times 4^{1.12}$. This comparison suggests that the proposed document driven methodology is at least as effective as those that the COCOMO estimates were based on. Although encouraging for the current study, this cannot really be taken as proof of improved productivity because the "experiment" only consists of one data point, the COCOMO model is a simple estimate, and the data used for COCOMO is not based specifically on SC programs.

4. Software requirements specification (SRS)

During the process of requirement gathering, the requirements are documented in a software requirements specification (SRS). The SRS is a document that clearly and precisely describes each of the essential requirements (functions, performance, constraints and quality attributes) of the software and external interfaces [44].

The experience of many software engineers suggests that software requirements can improve the software qualities listed in Section 2. For instance, *Reliability* is improved because the only way to judge reliability is to have requirements against which to compare the final product. Moreover, the process of writing software requirements, especially formal software requirements, improves the user's understanding of their problem and allows them to document the range of anticipated inputs that defines the usage profile for the software. Documentation of the anticipated inputs is important because restricting the allowed input is generally necessary to have a chance of effectively assessing *Reliability*, since many SC programs will be found to be unreliable if they are expected to handle all possible inputs. As a trivial example of this, most SC programs cannot handle values larger than the maximum floating point number on their system, even though this restriction is rarely explicitly stated. Moreover, there is usually a combination of input values that will cause an SC program to fail. These problematic input values could potentially be found near, but not exceeding, the extreme ends of the floating point number line. Besides protecting against failures, specifying the expected range for inputs provides an opportunity to improve *Performance*, over programs that do not exploit this information, because algorithms can be selected at the design stage that take advantage of the known usage profile.

In the proposed methodology, mathematical specification is used so that the requirements will be unambiguous, thus promoting *Verifiability* and facilitating the transition to design, implementation and test case derivation. Unambiguous requirements also assist with *Understandability* because a programmer will be able to quickly and accurately determine what the toolbox can and cannot do. Furthermore, a good SRS facilitates improving *Understandability* for both experts and non-experts, because it provides a structured presentation of all of the details relevant to a project. The improvement in *Understandability* for a range of levels of expertise facilitates communication between the different types of people that may be on a development team. With an SRS, clashes between software engineers and computational scientists, like the difficulties described in [35], can potentially be avoided. In the cases where the development team and the stakeholders only consist of experts, the documentation can be simplified because some user knowledge can be assumed. However, for this simplification to be valid, the assumed high level of expertise must be explicitly stated in the user characteristics section of the SRS.

Software requirements serve as a contract between developers and testers; therefore, the SRS promotes *Verifiability* by giving the testers something to verify against. Without unambiguous and validatable software requirements, it is impossible to test the software. *Reusability* is encouraged because software can only be reused if what the software does is known. This information is more easily obtained by reading the abstract SRS than by attempting to decipher all of the details in the concrete code. Those that are discouraged from reuse by the complexity of the implementation [8], should find the SRS a simpler starting point.

Fig. 2 shows the table of contents for PMGT's SRS. The template used to build the SRS is adapted from the descriptions given in [37,39,40]. Most of the sections in the current SRS are relatively standard for a requirements template [33,44]. However, the "Specific System Requirements" section deserves additional explanation, as this section is responsible for specifying the solution characteristics. This section presents the model used for a mesh by formally and informally specifying the goals the software should achieve, the assumptions made, the theoretical models involved and the data definitions employed. The "Specific System Requirements" section also lists the nonfunctional requirements, which are qualities that the software library should promote and facilitate when it is used by a given program.

Section "Specific System Requirements" begins by stating the goals of PMGT, where a goal is a functional objective that the

1. Reference Material: a) Table of Contents b) Table of Symbols c) Abbreviations and Acronyms d) Index of Requirements
2. Introduction: a) Purpose of the Document b) Scope of the Software Product c) Terminology Definition d) Organization of the Document
3. General System Description: a) System Context b) User Characteristics c) System Constraints
4. Specific System Description: i) Problem Description (including Goal statements) ii) Solution Characteristics Specification (including Theoretical models and Data Definitions) iii) Non-functional Requirements
5. Other System Issues: a) Open Issues b) Off-the-Shelf Solutions c) Waiting Room
6. Traceability Matrix
7. List of Possible Changes in the Requirements
8. Values of Auxiliary Constants

Fig. 2. Table of contents for SRS.

system under consideration should achieve [45]. The goals are stated abstractly to keep the requirements as general as possible. The goal will systematically be refined in the SRS and later through the design and implementation. One of the goals (**G1**) for PMGT is:

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

M^{IN} and M^{OUT} represent an input and an output mesh, respectively, and I represents instructions on how a mesh should be refined/coarsened. The instructions consist of marking cells that need to be refined or coarsened. The mathematical description given below will make this notion, and the model of a mesh, unambiguous.

After stating the goals for PMGT the assumptions are listed. The assumptions serve the vital purpose of reducing the scope of the software, so that the implementation will be feasible. If in the future the software library is extended, the extension will likely involve relaxing one or more assumptions. Explicitly including the assumptions makes future extension and the associated maintenance easier because the SRS traces the connection between assumptions, requirements and later the software modules. Documenting the assumptions improves the *Understandability* of the library, over libraries without documentation, because users explicitly know the library's limitations. Some example assumptions, using the numbering assigned in [49], are as follows:

> **A1**: PMGT focuses on a 2D domain.
> **A2**: The input and output meshes are bounded.
> **A4**: The input and output meshes are conformal.
> **A5**: The elements (cells) in the input and output meshes are triangles.

Theoretical models follow the assumptions in the SRS. The theoretical models refine the goals in two ways. First, they make the goals more concrete by applying the assumptions to the goals. Second, the theoretical models make the goals less ambiguous through the use of formal mathematics. A theoretical model that refines **G1** is as follows:

> **T1**: Refining mesh
> *Input*: M^{IN}: MeshT, I: RCinstructionT
> *Output*: M^{OUT}: MeshT such that Refined (M^{OUT}, M^{IN}, I)

This theoretical model states that the new mesh (M^{OUT}) is a refined version of the old mesh (M^{IN}), where the instructions I mark which cells need to be refined. As is typical for theoretical models, for **T1** to be complete, additional information is necessary. This information is given through what are termed data definitions. Just as for the theoretical model and for the goals, each data definition is given a unique number to assist with cross-referencing, thus facilitating *Maintainability* of the documentation. The types MeshT and RCinstructionT are defined in Fig. 3 and the data definition for the predicate Refined, using the numbering from [49], is as follows:

$\mathbf{D1}$: VertexT = tuple of $(x : \mathbb{R}, y : \mathbb{R})$
$\mathbf{D2}$: EdgeT = set of VertexT
$\mathbf{D4}$: CellT = set of VertexT
$\mathbf{D7}$: MeshT = set of CellT
$\mathbf{D20}$: InstructionT = {REFINE, COARSEN, NOCHANGE}
$\mathbf{D21}$: CellInstructionT = tuple of ($cell$: CellT, $instr$: InstructionT)
$\mathbf{D22}$: RCinstructionT = tuple of ($rORc$: InstructionT, $cInstr$: set of CellInstructionT)

Fig. 3. Some of the data types for modelling the requirements of PMGT.

> **D23**: Refined : MeshT × MeshT × RCinstructionT → $\mathbb{B}$
> Refined$(m', m, rc) \equiv (rc.rORc = \text{REFINE}) \wedge \text{ValidMesh}(m)$
> $\wedge \text{ValidMesh}(m') \wedge \text{CoveringUp}(m', m) \wedge (\#m' \geqslant \#m)$

Data definition **D23** uses × for the Cartesian product, $\mathbb{B}$ for the Boolean type, ∧ for logical AND, ≡ for defined equivalence and # for the cardinality of a set. The data definition shows that a refined mesh must be marked for refinement $(rc.rORc = \text{REFINE})$, that the new mesh and the old mesh must be valid meshes $(\text{ValidMesh}(m) \wedge \text{ValidMesh}(m'))$, that the new mesh must cover the same space as the old mesh $(\text{CoveringUp}(m', m))$, and that the new mesh should have as many or more cells than the old mesh $(\#m' \geqslant \#m)$. The reasoning behind this definition will be discussed further below. At this time however, additional data definitions need to be introduced to fully define **D23**. In particular, **D23** depends on several new types, which are listed in Fig. 3, using the reference numbers employed in [49] and the notation of [17].

As Fig. 3 shows a mesh is modelled as a set of cells and the cells themselves are modelled as a set of vertices, where each vertex is given as a tuple in $\mathbb{R}^2$, where $\mathbb{R}^2$ refers to a tuple of two real number $(\mathbb{R} \times \mathbb{R})$. The simple hierarchical relationship between vertices, cells and the mesh is convenient for expressing the requirements because it abstracts away many of the concrete data structure decisions that will later be necessary for any specific implementation. The data types in Fig. 3 also show how an entire mesh is either marked for refinement, coarsening, or no change and how individual cells are likewise marked.

The data definition of Refined (**D23**) also uses several additional predicates. In the interest of space, the predicate that returns true when both meshes cover the same area (CoveringUp) will not be reproduced here. However, as an illustrative example, the predicate that checks for mesh validity, ValidMesh (**D18**), is shown below.

> **D18**: ValidMesh: MeshT → $\mathbb{B}$
> $\text{ValidMesh}(m) \equiv (\forall e : \text{EdgeT} \mid e \in \text{Edges}(m) : \text{ValidEdge}(e)) \wedge$
> $(\forall c : \text{CellT} \mid c \in m : \text{ValidCell}(c)) \wedge \text{Bounded}(m) \wedge$
> $\text{Conformal}(m) \wedge$
> $\text{NoInteriorIntersect}(m)$

The notation used for **D18**, and for the other mathematical expressions in this paper, follows the expression building notation used in [13]. The notation can be explained by introducing a list of dummies x, a type t, a predicate R, an expression E, an operator $*$, and a predicate P. The notation $\{x : t \mid R : E\}$ represents a set of values that result from evaluating $E[x := v]$ in the state for each value v in t such that $R[x := v]$ holds in that state. Expression $(*x : t \mid R : P)$ denotes the application of operator $*$ to the values P for all x in t for

which range R is true. In **D18** $*$ is $\forall$, which means "for all". Besides understanding the notation, to interpret **D18** some additional information is needed as follows: Edges is a function that returns the edges in a mesh; ValidEdge is true if the edge is a line segment; ValidCell is true if the cell is a triangle; Bounded is true if the boundary edges form a closed polygon; Conformal is true if the intersection of any two cells is either a vertex, an edge or empty; and, NoInteriorIntersect is true if a point in space is inside only one cell of the mesh. The formal specification of these additional predicates can be found in [49].

As the above examples show, the mathematical specification of mesh refinement involves several new variables, types and functions. The potential complexity from managing this mathematical model motivates the following question: "Is the effort involved in documenting the requirements worthwhile?" The answer to this question is "Yes," if the requirements possess the qualities usually expected of good requirements; that is, the requirements need to be unambiguous (not open to interpretation), validatable (can be tested) and abstract (focus on "What" is required, not "How" to do it). The fact that the requirements are written formally suggests that they are unambiguous. Having unambiguous requirements is a precondition for having validatable requirements, but it is not a guarantee. To ensure that the requirements are validatable, the testability of the requirements was considered when they were written. In part this is why Refined (**D23**) has such a simple definition; if the meshes are valid and cover one another, the truth of Refined can be determined by simply counting the number of cells in the new and the old mesh. This is certainly something that can be easily tested, as described in Section 8. The final goal of abstract requirements facilitates achieving the software quality of *Portability*, since abstract requirements should not make specific statements about the operating environment. The selection of the operating environment is left as a design decision.

Although the goal of abstraction is important, there are times when this goal is sacrificed. For instance, in some cases a design decision is imposed for practical reasons. This is the case in the current project with the decision to improve meshing software via a software library, as opposed to a different structure, such as a stand alone program. The library choice was made because the library approach is common in mesh generation software, as it facilitates reuse of the code in many different contexts.

The abstractness of the requirements for PMGT are illustrated by the definition of Refined **D23**, which only requires that the new mesh have at least as many cells as the old mesh. With this definition almost any refinement algorithm would be admissible. Refined could be modified to provide additional information and still not explicitly identify the algorithm to be used. For instance, if the mesh application has strict mesh quality constraints, the requirement could state that the new mesh will not have any angles less than half the minimum angle in the old mesh. Another condition that could be added to Refined would be to explicitly state that the refined cells are those that are explicitly marked for refinement; that is, the refined cells will be those for which $ci.instr = \text{REFINE}$. To achieve this, the following predicate $(\text{CellsRefined}(M^{OUT}, M^{IN}, I))$, which uses the data types defined in Fig. 3, could be conjuncted with data definition **D23**.

$$
\begin{aligned}
&\text{CellsRefined} : \text{MeshT} \times \text{MeshT} \times \text{RCinstructionT} \rightarrow \mathbb{B}\\
&\text{CellsRefined}(m', m, rc) \equiv \forall ci : \text{CellInstructionT} \mid\\
&\quad ci \in rc.cInstr \wedge ci.instr = \text{REFINE}:\\
&\quad \exists m'' : \text{MeshT} \mid m'' \subset m' : \text{ValidMesh}(m'') \wedge\\
&\qquad \text{ValidMesh}(\{ci.cell\}) \wedge\\
&\quad \text{CoveringUp}(m'', \{\text{ci.cell}\}) \wedge \#m'' \geqslant \#\{\text{ci.cell}\}
\end{aligned}
$$

The above predicate, which uses $\exists$ for existential quantification, requires that a cell marked for refinement in m be decomposed into multiple cells, which cover the same area as the original cell, in m'. The restriction that the refinement of a cell covers the same space as the original cell is restrictive, which may be inappropriate in some contexts. The advantage of an SRS is that it facilitates the discussion of the appropriateness of this restriction in advance of the implementation.

The theoretical models in the SRS can be further refined with additional information documented in the form of functional and nonfunctional requirements. Some example functional requirements for PMGT are given in Table 1. Each requirement is given a unique label, identified by F?, where ? is a natural number. In the full documentation of the requirements [49] additional information is given for each requirement, including the source of the requirement, a list of related data definitions and theoretical models and a record of the history of changes to the requirement.

Table 1 also lists several sample nonfunctional requirements (NFRs), which are uniquely identified using numbers prefixed with N. The challenge for the nonfunctional requirements is to state them in such a way that they are validatable. It is inadequate to simply have a performance requirement that the software be "fast", since this is ambiguous and cannot be tested. The approach taken for NFRs to address this problem is to state them as relative requirements between competing algorithms and different software packages [37]. For instance, in Table 1 performance and understandability requirements are stated relative to the performance and understandability of the mesh generation library AOMD [31]. To make the nonfunctional requirements validatable (testable) they have to be specific. In the current case, the specific test problem, called Example D in the table, is given in Appendix D of [49].

A traceability matrix is used in the SRS to show the relationship between goals, assumptions, theoretical models, data definitions, functional requirements and nonfunctional requirements. A portion of the matrix from [49] is shown in Table 2. This matrix assists with the *Maintainability* of PMGT by documenting the impact of changes within the SRS document itself. For instance, if an assumption should change, then the SRS specifies which portions of the document need to be updated. An example of this would be changing the assumption that the domain is 2D (**A1**) to an assumption that it is 3D. Table 2 shows that changing **A1** means that the assumption that the elements are triangles (**A5**) would also have to be changed; the elements will have to become shapes with a 3D topology, such as tetrahedra or hexahedra. Changing the dimension of the domain would also mean that the data definitions for VertexT (**D1**) and CellT (**D4**) would need to be modified. In the case of the vertex information, the change would be to incorporate another dimension for the coordinates. For the cell definition, the change to a 3D topology would mean the cells are no longer represented by just three vertices, so a set of VertexT would no longer implicitly carry the connectivity information. A different data type, possibly using a sequence of VertexT, would need to be adopted.

Documenting traceability between portions of the SRS and, as shown in the next section, between the requirements and the design, are very important to successful SC software development. This is especially true during the initial iterative phases of discovery and experimentation, when changes will likely be frequent. As mentioned previously, although the documents follow a rational sequence, the development itself will involve considerable iteration. Some sections of the SRS can be filled in initially, such as the goal section, user characteristics, and scope section, while others will be added over time. As changes are made, for instance to the assumptions, the effect of these changes on the SRS and on subsequent documents can be traced.

Table 1
Sample requirements.

Num	Label	Description
F1	RefiningMesh	PMGT should have capabilities for refining an existing mesh
F2	CoarseningMesh	PMGT should have capabilities for coarsening an existing mesh
F3	RefiningOrCoarsening	PMGT can either refine or coarsen a given mesh or coarsen a mesh, but not both at the same time
F4	MeshType	The mesh generated by PMGT is unstructured
F11	VertexUniqueID	Each vertex in the output file has a unique identifier
F12	ElmUniqueID	Each element in the output file has a unique identifier
F13	ElmTopology	The topology of an element in the output file is given by the connectivity of its set of vertices
F14	OutElmOrder	The element information in output files is listed in ascending order
F15	OutVertexOrder	The vertex information, such as the coordinates, in output files is listed in ascending order
N1	Performance	Refining/coarsening a mesh using multiple processors should be faster than when using a single processor. In addition, the performance of PMGT should be comparable with that of similar applications. Specifically, the execution time to mesh Example D should be no longer than the time taken by AOMD to solve the same problem
N6	Understandability	Users with the background specified in the SRS should be able to set-up Example D at least as quickly as an AOMD user could

Table 2
Traceability matrix between goals, assumptions, theoretical models and data definitions.

	G1	A1	A2	A4	A5
A1	✓	✓			
A2	✓		✓		
A4	✓			✓	
A5	✓	✓			✓
D1	✓	✓			
D2	✓				
D4	✓	✓			✓
D7	✓				
D18	✓				
D20	✓				
D21	✓				
D22	✓				
D23	✓				
T1	✓	✓			

5. Module guide (MG)

According to the waterfall model for documenting PMGT, as shown in Fig. 1, the software design documents follow the SRS. Software design involves decomposing the software into modules, where the definition of a module adopted here is a "work assignment" [24]. A summary of what the modules are intended to do and the relationship between them is provided in the Module Guide (MG) document.

The principle applied to the design is *information hiding*. According to this principle, system details that are likely to change independently should be hidden in different modules [27]. The information hiding principle allows both designers and maintainers to easily identify the parts of the software that they want to consider without needing to know irrelevant details. The *Maintainability* of the software is thus improved over software where changes to the implementation require modifications in many locations. PMGT hides details of the underlying operating system, which is a common strategy in software design for improving *Portability*. Table 3 summarizes the modules of PMGT along with the information that they hide (secrets).

The first step in the system architectural design, as summarized in the MG, is identifying the anticipated changes, such as those related to data structures, algorithms and portability. The anticipated changes guide the design so that it can explicitly account for likely potential future modifications. Ideally, the anticipated changes should be independent, so that each change can be hidden within one module. When a change occurs, only the module that hides the change needs to be modified. Several example anticipated changes for PMGT, using the numbering from [49], are as follows:

AC2: The data structure and algorithms for implementing the interface between a file and the software
AC7: The mechanisms for validating the input and output meshes
AC11: The data structure of a mesh
AC12: The algorithms for refining a mesh
AC14: The shape of the cells in the mesh (initially PMGT is for triangular meshes)

Table 4 shows the traceability between the above anticipated changes and the associated modules of PMGT. The names of the modules are show in the first row. The first column, "File Read Write" module, shows how the design facilitates the portability of PMGT. On a different operating system, files will be handled differently, so there is a need for a module to hide the data structures and algorithms used for file handling. Fortunately most

Table 3
Modules for PMGT.

Num	Label	Secret
M1	Virtual memory	The hardware addressing methods for data and instructions in real memory
M2	File read write	The data structure and algorithms for implementing the interface between the file and the system
M3	Keyboard input	The data structure and algorithms for implementing the interface between the keyboard and the system
M4	Screen display	The data structure and algorithms to display graphics and text on the screen
M5	Input format	The format and structure of the initial input mesh
M6	Output format	The format and structure of the output mesh
M7	Service	The algorithm for validating meshes
M8	Vertex	The data structure of a vertex
M9	Edge	The data structure of an edge
M10	Cell	The data structure of a cell
M11	Mesh	The data structure of a mesh
M12	Refining	Algorithms for refining a mesh
M13	Coarsening	Algorithms for coarsening a mesh

Table 4
Excerpt from a traceability matrix between modules and anticipated changes.

	File read write (**M2**)	Service (**M7**)	Cell (**M10**)	Mesh (**M11**)	Refining (**M12**)
AC2	✓				
AC7		✓			
AC11				✓	
AC12					✓
AC14		✓	✓		

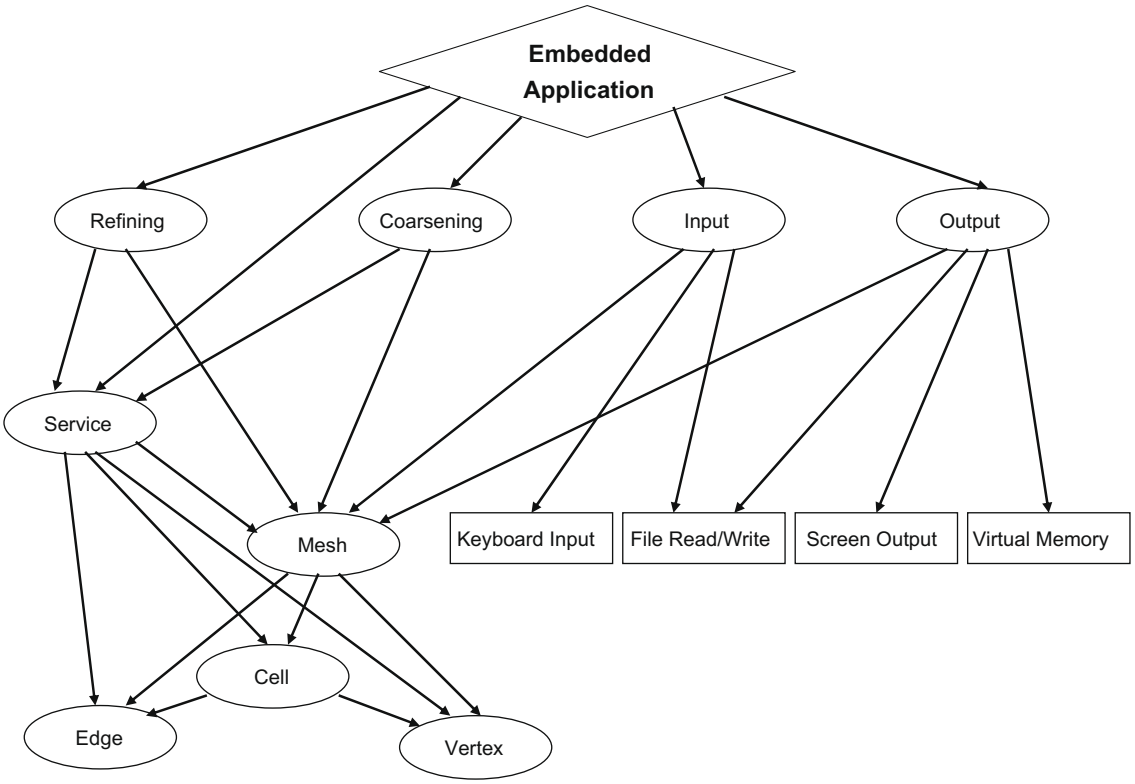

Fig. 4. Uses hierarchy.

modern programming languages already hide the file system, so it is unlikely that this module will have to be explicitly implemented. As Table 4 shows, in most cases there is a one to one correspondence between anticipated changes and modules. For instance, if the data structure for representing a mesh should change (**AC11**), the only module that will need to be modified is the Mesh module. **AC14** is related to two modules because the service module and the cell module need to know the number of vertices for a cell. Table 4 shows that the algorithm and the data structure have been uncoupled, which facilitates changes to the algorithm. All access to the mesh data will be through a stable standardized interface, which is discussed in the Section 6.

Unlikely changes are also listed in the MG. If one of these changes occurs, the design of PMGT makes no obligation that adapting to this change will only require a small modification. Listing unlikely changes helps one set realistic goals for *Maintainability*. Some unlikely changes for PMGT include:

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

To illustrate the influence of unlikely changes, one can consider changing **UC6**, the representation of an edge. Changing the representation of an edge will affect all of the data structure modules, since the implementation assumes that an edge is represented by two vertices. If an edge is instead represented by two cells, then this change cannot be isolated to a single module. Another example of the influence of an unlikely change is to consider a change in the coordinate system (**UC8**). Although considered unlikely in the current design, the design proposed by [10] considers a change in the coordinate system as a likely change. If one compares the current design to [10] one will observe that [10] is more general, but at the expense of greater abstraction and potentially lessened *Understandability*.

Software design includes specifying the relationship between modules. The use relation for PMGT is shown in Fig. 4. For two modules A and B, A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification [26]. The modules shown in rectangles, which correspond with the hardware hiding modules, are assumed to already be implemented in the programming environment. Fig. 4 is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system. This potentially improves the *Verifiability* and *Reusability* of PMGT. For example, the mesh module (together with the vertex module, edge module, and cell module) is a subset of the

Table 5
Excerpt from a traceability matrix between modules and requirements.

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13
F1												✓	
F2													✓
F3												✓	✓
F4									✓	✓	✓	✓	✓
F11						✓							
F12						✓							
F13						✓							
F14						✓							
F15						✓							
N1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
N6			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

system. The design and the implementation of this subset can be reused since it does not use other modules. Modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels, thus the *Maintainability* of PMGT is encouraged. The design is easily understood due to its simplicity, which promotes the *Understandability* of PMGT.

The MG facilitates *Maintainability* of the software because it shows the traceability between requirements and design, which allows one to determine which software modules need to be changed when there is a change in the requirements. Table 5 shows how changes in some functional and nonfunctional requirements made to the modules in PMGT's design. For instance, if a modification is made to any of the requirements that deal with the output, such as requirements **F11**–**F15**, then the output format hiding module will have to be modified. Table 5 reinforces the fact that changes in the output format are isolated to one module. The rows full of check marks for the NFRs in Table 5 highlight the fact that NFRs are related to qualities of the overall system; thus, changes to NFRs cannot be isolated to a small number of modules. The traceability between requirements and design assists with an iterative development process because the effect of change is readily apparent.

6. Module interface specification (MIS)

The MG does not provide enough information for each module to be developed independently. The syntax and semantics of the access routines for each module are still needed. The detailed design of PMGT is described in the Module Interface Specification (MIS) document [49]. The MIS is less abstract than the architectural design in the last section. However, it is still abstract because it describes what the module will do, but not how to do it. The MIS encourages *Reusability* because it assists with the *Understandability* of the program interface by clearly specifying the meaning of access program parameters, since an unclear description of the parameters is one reason for not reusing libraries [8].

The template used to document the MIS for each module is a modified version of the templates presented in [12,17]. According to the adopted template, each module is a finite state machine (FSM). A simple formalism [17] for an FSM is a tuple (S, s_0, I, O, T, E), where S is a set of states, s_0 is an initial state $(s_0 \in S)$, I is the set of inputs, O is the set of outputs, T is the transition function $(T : S \times I \rightarrow S)$ and E is the output function $(E : S \times I \rightarrow O)$. The FSM model and the associated documentation provides an unambiguous statement of the module's services and its assumptions about the outside environment. An FSM specification of a module clarifies the documentation of the module and thus potentially improves *Understandability*, *Reusability*, *Maintainability* and *Verifiability*. The Mesh module will be used to illustrate how the syntax (Fig. 5) and semantics (Fig. 6) of the module state machine are presented.

1. **Imported Data Types**
   Uses *Vertex Module* Imports VertexT
   Uses *Edge Module* Imports EdgeT
   Uses *Cell Module* Imports CellT

2. **Exported Data Types**
   MeshT = set of CellT

3. **Exported Access Programs**

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

Fig. 5. Syntax of the mesh module.

The syntax of the MIS for each module documents the imported and exported data types and the exported access programs. As Fig. 5 shows, the Mesh module uses the types $\mathbb{B}$ (Boolean) and $\mathbb{N}$ (natural number), which are both assumed to be defined by whatever programming language will eventually be used for the implementation. The Mesh module also uses three types defined by the specification: VertexT, EdgeT and CellT. The name of the module that defines the imported type is given so that the reader can easily navigate the document and find any details that they may be seeking. The syntax for the mesh module states that MeshT is an exported type, which means that other modules may import this type to use in their specification. It is worth noting that the importing and exporting of types are used here for specification purposes; the actual implementation does not necessarily follow the same pattern. In particular, the implementation of a given module will often use modules that are not listed in the MIS. This is a consequence of the increase in the amount of detail associated with moving from an abstract interface to a concrete implementation.

Fig. 5 lists the access programs for the Mesh module. These programs are the only interface to the mesh module, analogous to public methods in an object oriented design. This interface must remain stable, but the internal design decisions for the Mesh module can be changed. For each access program, the input and output types are listed along with the exceptions, which are generated when an undesired condition occurs. Although the syntax section of the MIS shows useful information, a programmer also needs to know how the state of the module and its output will change depending on the current state and the provided input. This

4. **State Variables**
   m: set of CellT
5. **Invariant**
   $\#m \geq 0$
6. **Assumptions**
   initMesh() is called before any other access routines.
7. **Access Program Semantics**
   - initMesh()
     - **Transition**
       $m := \emptyset$
   - addCell(c: CellT)
     - **Exception**
       $c \in m \Longrightarrow$ CellExist
     - **Transition**
       $m := m \cup \{c\}$
   - onEdge(v: VertexT, e: EdgeT)
     - **Description**
       Returns true if a vertex v is on the line segment between two vertices (exclusive) of the edge e.
     - **Output**
       $\exists\ v_1, v_2$: VertexT $\mid v_1 \in e \wedge v_2 \in e \wedge v_1 \neq v_2 \wedge v \neq v_1 \wedge v \neq v_2 :$
       $(v_1.x < v.x \leq v_2.x \wedge (v.y - v_1.y)/(v.x - v_1.x) = (v_2.y - v_1.y)/(v_2.x - v_1.x))$
   - belongToCell(e: EdgeT, c: CellT)
     - **Description**
       Returns true if an edge e belongs to a cell c.
     - **Output**
       $\forall\ v$: VertexT $\mid v \in e : v \in c$
   - edges()
     - **Description**
       Returns a set of all edges of the mesh
     - **Output**
       $\{v_1, v_2 :$ VertexT $\mid (\forall c$: CellT $\mid c \in m : v_1 \in c \wedge v_2 \in c \wedge v_1 \neq v_2) : \{v_1, v_2\}\}$
   - boundaryEdges()
     - **Description**
       Returns the set of boundary edges of the mesh
     - **Output**
       $\{b$: EdgeT $\mid b \in$ edges() $\wedge$ (#$\{c$: CellT $\mid c \in m \wedge$ belongToCell(b, c): $c\} = 1$): $b\}$

Fig. 6. Excerpt from semantics of the mesh module.

information is provided in the semantics section, for which an excerpt is provided in Fig. 6.

Fig. 6 shows that the state variable for the Mesh module is m, a set of cells. The documentation also identifies the invariant that the number of cells will always be zero or greater ($\#m \geqslant 0$). This invariant potentially improves the quality of *Verifiability* of the documentation and code because the documentation and code may be inspected to verify that if the state invariant holds at the beginning of an access program call, it will still hold at the ending of that call. Section 6 of the Mesh module's MIS shows that the implementation is allowed to assume that the programmer will not call any access programs until after initMesh() has been called. That is, initializing the Mesh module is the responsibility of the programmer. This decision was made to avoid the necessity of adding state information on initialization, the need to frequently check this information and the need to provide an associated exception.

For a typical usage of the Mesh module the Mesh is first initialized with the state transition $m := \emptyset$, where $\emptyset$ is the null set. After this, cells are added to the set of cells in the mesh using addCell(), which employs the transition $m := m \cup \{c\}$, where $\cup$ is set union. In the event that a cell is already in the mesh, an exception is generated ($c \in m \Rightarrow$ CellExist) because a set should only store one instance of each of its constituent elements. Once the mesh is built there are several derived quantities and predicates of interest. For instance, a query can be made to see whether a given vertex is on a given edge. This query is made using similar triangles, with the resulting predicate shown in the output field of the onEdge() access program in Fig. 6. Another example of the use of the Mesh module is generating output of the sets of edges and of edges on the mesh boundary. These sets are found using the edges() and boundaryEdges() access programs, respectively. Although the output field for boundaryEdges() looks complicated, the mathematics simply state that the returned value will be the set of edges where each edge belongs to only one cell in the mesh, since this corresponds with the set of boundary edges.

7. Implementation (code)

The system implementation is the transformation of the design to a work product. The implementation phase is very important in the software development life cycle because it produces an executable version of the system. In many SC projects the implementation is the only phase of the software development life cycle that receives much attention. However, even when the implementation is the sole component, in many cases there is still room for quality improvement. For instance, one reason for poor quality SC software is that the software is often written by scientist and most scientists have simply never been shown how to program efficiently [48]. In

particular, support technology that can assist with software development, such as version control software, is often neglected, even when it can potentially improve quality.

The implementation is the final step of the refinement from abstract to concrete. The major decisions relating to the implementation of PMGT are the data structures and algorithms. Rather than develop a new data structure or algorithm for mesh generation consideration was given to selecting (with minor modifications if necessary) existing data structures and algorithms to fit the scope of PMGT and to improve the qualities of PMGT. An edge-based data structure was selected for PMGT because of its flexibility (*Maintainability*), which is especially important if the anticipated change of cell shape (**AC14**) should be made in the future. The data structure for PMGT is based on the halfedge data structure [22].

For the serial version of PMGT two algorithms were implemented: (i) a point insertion algorithm; and, (ii) the longest edge bisection algorithm. The point insertion algorithm adds a point at the centroid of an existing element and refines the element into three triangles. The longest edge bisection algorithm is a modified version of the algorithm found in [32]. For each triangle to be refined, the mesh is updated based on a list of successive neighbour triangles, where the members of the list have longest-edges greater than the longest edge of the preceding triangle in the list. The longest edge bisection algorithm produces meshes that are generally of better quality than those produced by the point insertion algorithm.

SE technology is used through the implementation of PMGT to improve the software quality. This technology can improve the overall quality of the software development process, as well as the software product. Version control is used for the entire development of PMGT. One of the advantages of version control is that it helps diagnose errors by facilitating comparison of a new incorrect version of the code to an old correct version of the code. Therefore, version control assists with improving the *Maintainability* of the software. In addition to version control, a makefile was employed, where the makefile manages the build process, which can be error prone when done manually; therefore, adopting a makefile benefits the *Reliability* of PMGT. Finally, namespaces were used in the C++ source code to avoid name conflict and unnecessary access of protected data and routines; this decision promotes the *Reliability* and *Understandability* of PMGT.

PMGT will execute on any Linux/Unix/Mac operating system with a g++ compiler and the MPI library. For the serial version the MPI library is not necessary; therefore, the serial version can be easily adapted to the Windows OS. Implementing PMGT for multiple environments improves its *Portability*.

8. Software testing

Testing is often not emphasized in SC and when it is conducted "scientists use testing to show that their theory is correct, not that the software does not work" [34]. That is, rather than verify that the implementation of the theory is correct, the first assumption made by developers is that any errors are in the theory itself. Given the inevitability of coding errors, this confidence in the implementation is often misplaced. Part of the reason for this misplaced confidence may be that a systematic process is rarely followed for SC software development, so a systematic process is also not followed for testing. As an example of this possibility, planning systematic unit tests is difficult if one has not previously designed the software as a set of units (modules). Another part of the explanation for limited testing, may be that SE methodologies for testing do not specifically address the most significant challenge in SC testing: the challenge of finding meaningful test cases with known solutions. As mentioned in Section 2, one of the important characteristics of SC software is that it is difficult to find true solutions for most SC problems.

Two approaches are used in the SVTR to deal with the unknown solution challenge: (i) parallel testing; and, (ii) verification of derived properties or characteristics. Parallel testing involves testing independent implementations of the same computational model against one another. This is what Hatton [14,15] does to find errors in seismic data processing programs. Parallel testing is also the motivation for phrasing the NFRs for PMGT as relative comparisons to AOMD (Table 1). If parallel testing of two programs shows disagreement, then one knows at least one of the two packages is incorrect. For the second approach, the verification of derived properties, the goal is changed from finding the true solutions, to finding properties that the true solution is known to have. Some example properties from physical modelling are the requirements for equilibrium and for conservation of mass. Although the notion of a "true" solution is not entirely clear for a mesh, since small changes can be made in the location of some of the interior nodes and the mesh is still valid, derived properties of a correct mesh can still be determined.

One way to test the *Reliability* of PMGT is to see whether the output mesh is a refined or coarsened version of the input mesh. According to the SRS, the characteristics of a refined and coarsened mesh relate to the data definition of Refined (**D23**) and Coarsened, respectively. In both definitions, the output mesh needs to be a valid mesh and the input mesh and output mesh need to cover each other. The predicates for a valid mesh and a covering up are defined in the SRS. In addition, other requirements that are common to all meshes should be met. For instance the requirement should be reached that the mesh satisfies the Euler Equation, $nc + nv - ne = 1$, where nc is the number of cells, nv is the number of vertices, and ne is the number of edges [11].

To improve the *Understandability* of PMGT, the correctness tests are automated. A significant advantage of automated testing is that it allows for regression testing, so that when changes are made to the software, previous test cases can easily be redone. The automated correctness validation test requirements (ACVTRs) of PMGT are as follows:

1. The area of each element is greater than zero.
2. The boundary of the mesh is closed.
3. The mesh is conformal.
4. The intersection of any two elements is empty.
5. The input mesh and output mesh cover each other.
6. The length of each edge is greater than zero.
7. The vertices of an element are listed in a counterclockwise order.
8. The output mesh conforms to the Euler Equation.

Since the output mesh can be displayed on the screen, the output meshes can also be visually checked to ensure that no vertex is outside the input domain, no vertex is inside a cell, no dangling points or edges exist, all cells are connected and the mesh is conformal. Some of these requirements overlap with the ACVTRs. This redundancy is an advantage for promoting *Reliability* because the odds of misinterpreting two independent tests are very small. For the final version of PMGT all test cases were passed.

An important purpose of testing in SC is to describe the quality of the software. For instance, the *Performance* can be estimated through experimentation with the code. The execution time for refining a mesh using PMGT was measured using the serial version and the parallel version with different number of processors. The results showed a super linear speedup with an increasing number of processors. The dramatic speedup is explained by the fact that the parallel algorithm only implemented the point insertion algorithm (Section 7), which allows for an embarrassingly parallel

algorithm because no communication is necessary between the cells of the mesh. The fact that the speedup is super linear is probably due to the cache effect. That is, when the numbers of processors increases, the size of accumulated caches from different processors increases as well. With the larger accumulated cache size, more data can fit into the caches and the memory access time reduces dramatically, which causes the extra speedup in addition to the speedup from parallelization of the computation.

The emphasis above was on system tests assessing *Reliability* and *Performance*, since it is the final system we are most interested in and these two qualities are generally considered to be the most important in SC. Moreover, we focused on the unknown solution challenge because it is a defining characteristic of SC. Testing can also be done using more conventional SE practises. For instance black box unit tests can be constructed based on the specifications given in the MIS. In addition, white box tests inspired directly from the code can be devised. These tests can also generate a profile that quantifies such metrics as statement and condition coverage.

As for the SRS and MG, traceability is an important component of the SVTR. A matrix is provided in [49] to show which test cases exercise which requirements. By inspecting the matrix, it is possible to see that all of the functional requirements of SVTR are covered by the test cases. The matrix relating test cases and requirements facilitates a nonlinear process for developing SC software because it allows one to concurrently work on the SRS and SVTR. Test cases complement the requirements, and in some sense can be thought of as requirements, so it makes sense to document them at the same time. A matrix is also provided [49] to show which modules are needed by each of the test cases. This matrix can be inspected to ensure that each module is tested by at least one test case. Moreover, if a test case fails, the traceability matrix shows which modules need to be investigated to track down the error.

9. Concluding remarks

This paper proposes adapting SE methodologies to improve the quality of SC applications by illustrating the design and documentation of PMGT. As an example of a quality improvement, the quality of *Reliability* is potentially improved by having explicit traceability between requirements, design, implementation and testing, since the traceability can be inspected to improve the verification of completeness and consistency. *Reliability* may also be improved because having an unambiguous mathematical specification of the requirements and the module interfaces allows one to judge correctness. Although the correct mesh is not known a priori, the proposed methodology can still promote *Reliability* and measure *Accuracy* through the automatic test cases that verify properties of a correct solution, rather than the solution itself. In addition to the validity of the solution, testing also measured the quality of *Performance* and demonstrated that the parallel version of the program met the nonfunctional requirement of being faster than the serial version.

SE methodologies also promote the related qualities of *Understandability*, *Reusability*, *Verifiability* and *Maintainability*. The principle of information hiding and the emphasis on identifying anticipated changes means that programmers will not be distracted by unnecessary implementation details and that the design will accommodate future growth. For instance, information hiding assists with the *Portability* of the PMGT library. The uses hierarchy between modules also helps because it shows the relationship between modules and the useful subsets of the software library. Finally the previously mentioned traceability assists with maintenance tasks (*Maintainability*) because the connections between design documents is explicit.

Acknowledgements

The financial support of SHARCNET and the Natural Sciences and Engineering Research Council (NSERC) are gratefully acknowledged.

References

[1] Ardis Mark, Weiss David M. Defining families: the commonality analysis. In: Proceedings of the nineteenth international conference on software engineering. ACM, Inc.; 1997. p. 649–50.
[2] Cecilia Bastarrica M, Hitschfeld-Kahler Nancy, Rossel Pedro O. Product line architecture for a family of meshing tools. In: Morisio Maurizio, editor. ICSR. Lecture notes in computer science, vol. 4039. Springer; 2006. p. 403–6.
[3] Bastarrica María Cecilia, Hischfeld-Kahler Nancy. Designing a product family of meshing tools. Adv Eng Software 2005:1–10.
[4] Boehm Barry. Software engineering economics. Englewood Cliffs (New Jersey): Prentice Hall; 1981.
[5] Clements Paul, Northrop Linda M. Software product lines: practices and patterns. Boston (MA, USA): Addison-Wesley Longman Publishing Co., Inc.; 2002.
[6] Cuka David A, Weiss David M. Engineering domains: executable commands as an example. In: International conference on software reuse. IEEE Computer Society; 1998. p. 26–34.
[7] Dijkstra EW. Structured programming, chapter notes on structured programming. London: Academic Press; 1972.
[8] Dubois Paul F. Designing scientific components. Comput Sci Eng 2002;4(5):84–90.
[9] Einarsson Bo, Boisvert Ronald, Chaitin-Chatelin Françoise, Cools Ronald, Douglas Craig, Dritz Kenneth, et al. Accuracy and reliability in scientific computing. Number 0-89871-584-9 in software-environments-tools. Philadelphia (PA): SIAM; 2005.
[10] ElSheikh Ahmed H, Spencer Smith W, Chidiac Samir E. Semi-formal design of reliable mesh generation systems. Adv Eng Software 2004;35(12):827–41.
[11] Frey Pascal Jean, George Paul-Louis. Mesh generation application to finite elements. Hermes Science Europe Ltd.; 2000.
[12] Ghezzi Carlo, Jazayeri Mehdi, Mandrioli Dino. Fundamentals of software engineering. 2nd ed. Upper Saddle River (NJ, USA): Prentice Hall; 2003.
[13] Gries David, Schneider Fred B. A logical approach to discrete math. Springer-Verlag Inc.; 1993.
[14] Hatton Les. The chimera of software quality. Computer 2007;40(8).
[15] Hatton Les, Roberts Andy. How accurate is scientific software? IEEE Trans Software Eng 1994;20(10):785–97.
[16] Heitmeyer Constance. Software cost reduction. In: Marciniak JJ, editor. Encyclopedia of software engineering. John Wiley & Sons, Inc.; 2002.
[17] Hoffman Daniel M, Strooper Paul A. Software design, automated testing, and maintenance: a practical approach. International Thomson Computer Press; 1995.
[18] Kelly Diane F. A software chasm: software engineering and scientific computing. IEEE Software 2007;24(6):119–20.
[19] Kelly Diane F, Sanders Rebecca. Assessing the quality of scientific software. In: Proceedings of the first international workshop on software engineering for computational science and engineering (SECSE 2008), Leipzig, Germany. In conjunction with the 30th international conference on software engineering (ICSE); 2008.
[20] McCall J, Richards P, Walters G. Factors in software quality. NTIS AD-A049-014, 015, 055; November 1977.
[21] Oliveira Suely, Stewart David E. Writing scientific software: a guide to good style. New York (NY, USA): Cambridge University Press; 2006.
[22] OpenMesh. Openmesh, computer graphics and multimedia group, Rheinisch-Westfälische Technische Hochschule Aachen, <http://www.openmesh.org/>; 2006.
[23] Owen Steven J. A survey of unstructured mesh generation technology. In: Proceedings 7th international meshing roundtable, Dearborn, MI; October 1998.
[24] Parnas DL. A technique for the specification of software modules with examples. CACM 1972;15(5):330–6.
[25] Parnas David. On the design and development of program families. IEEE Trans Software Eng 1976;SE-2(1):1–9.
[26] Parnas David L. Designing software for ease of extension and contraction. IEEE Trans Software Eng 1979(March):128–38.
[27] Parnas David L. Some software engineering principles. INFOR, Can J Oper Res Inform Process 1984;22(4):303–16.
[28] Parnas David L, Clements PC. A rational design process: how and why to fake it. IEEE Trans Software Eng 1986;12(2):251–7.
[29] Parnas David Lorge, Asmis GJK, Madey J. Assessment of safety-critical software in nuclear power plants. Nucl Safety 1991;32(2):189–98.
[30] Pohl K, Böckle G, van der Linden F. Software product line engineering: foundations principles and techniques. Springer-Verlag; 2005.
[31] Remacle Jean-Fancois, Shephard Mark S. An algorithm oriented mesh database. Int J Numer Methods Eng 2003;58:349–74.
[32] Rivara Maria-Celilia. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulation. Int J Numer Methods Eng 1997;40:3313–24.

[33] Robertson Suzanne, Robertson James. Mastering the requirements process. ACM Press; 1999.
[34] Sanders Rebecca, Kelly Diane. Dealing with risk in scientific software development. IEEE Software 2008;4:21–8. July/August.
[35] Segal Judith. Models of scientific software development. In: Proceedings of the first international workshop on software engineering for computational science and engineering (SECSE 2008), Leipzig, Germany. In conjunction with the 30th international conference on software engineering (ICSE); 2008.
[36] Siegel Stephen F, Mironova Anastasia, Avrunin George S, Clarke Lori A. Using model checking with symbolic execution to verify parallel numerical programs. In: International symposium on software testing and analysis (ISSTA), Portland (ME); 2006.
[37] Spencer Smith W. Systematic development of requirements documentation for general purpose scientific computing software. In: Proceedings of the 14th IEEE international requirements engineering conference, RE 2006, Minneapolis/ St. Paul, Minnesota; 2006. p. 209–18.
[38] Spencer Smith W, Chen Chien-Hsien. Commonality analysis for mesh generating systems. Technical report CAS-04-10-SS, McMaster University, Department of Computing and Software; 2004.
[39] Spencer Smith W, Lai Lei. A new requirements template for scientific computing. In: Ralyté J, Agerfalk P, Kraiem N, editors. Proceedings of the first international workshop on situational requirements engineering processes – methods, techniques and tools to support situation-specific requirements engineering processes, SREP'05, Paris, France, 2005. In conjunction with 13th IEEE international requirements engineering conference; 2005. p. 107–21.
[40] Spencer Smith W, Lai Lei, Khedri Ridha. Requirements analysis for engineering computation: a systematic approach for improving software reliability. Reliable Comput 2007;13:83–107. Special Issue on Reliable Engineering Computation.
[41] Spencer Smith W, McCutchan John, Cao Fang. Program families in scientific computing. In: Sprinkle Jonathan, Gray Jeff, Rossi Matti, Tolvanen Juha-Pekka, editors. 7th OOPSLA workshop on domain specific modelling (DSM'07), Montréal, Québec; 2007. p. 39–47.
[42] Squires S, Van de Vanter ML, Votta LG. Software productivity research in high performance computing. CT Watch Quart 2006(November):52–61.
[43] Tang Jin. Developing scientific computing software: current processes and future directions. Master's thesis, McMaster University, Hamilton, ON; 2008.
[44] Thayer RH, Dorfman M, editors. IEEE recommended practice for software requirements specifications. Washington (DC, USA): IEEE Computer Society; 2000.
[45] van Lamsweerde Axel. Goal-oriented requirements engineering: a guided tour. In: Proceedings of the 5th IEEE international symposium on requirements engineering IEEE. Washington (DC, USA): IEEE Computer Society; 2001. p. 249–63 [August].
[46] Weiss D, Lai CTR. Software product line engineering. Addison-Wesley; 1999.
[47] Weiss David M. Commonality analysis: a systematic process for defining families. Lect Notes Comp Sci 1998;1429:214–22.
[48] Wilson Gregory V. Where's the real bottleneck in scientific computing? Scientists would do well to pick some tools widely used in the software industry. Am Sci 2006;94(1).
[49] Yu Wen. A document driven methodology for improving the quality of a parallel mesh generation toolbox. Master's thesis, McMaster University; 2007.