FASCS: A FAMILY APPROACH FOR DEVELOPING SC SOFTWARE

FASCS: A FAMILY APPROACH FOR DEVELOPING SCIENTIFIC COMPUTING SOFTWARE

By

WEN YU, M.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Doctor of Philosophy

McMaster University

DOCTOR OF PHILOSOPHY (2011) McMaster University

COMPUTING AND SOFTWARE Hamilton, Ontario

TITLE: FASCS: A Family Approach for Developing Scientific Computing Software

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

SUPERVISOR: Dr. Spencer Smith

NUMBER OF PAGERS: xx, 192

Abstract

Scientific Computing (SC) software has had considerable success in achieving improvements in the quality factors of accuracy, precision and efficiency. However other software quality factors, such as reusability, maintainability, reliability and usability are often neglected. This thesis proposes a new methodology, Family Approach for developing Scientific Computing Software (FASCS), to improve the overall quality of SC software. In particular, the aim is to benefit the development of professional end user developed SC programs.

FASCS is the first methodology to apply a family approach to develop SC software, where all stages in both the domain engineering phase and the application engineering phase are included. In addition, the challenges for SC software and the characteristics of professional end user developers are also considered. A proof of concept program family, FFEMP, which can solve elasticity problems in solid mechanics using the Finite Element Method (FEM), is developed to illustrate how the proposed methodology can be used.

Part of FASCS is a new methodology for systematically eliciting, analyzing and documenting common and variable requirements for a program family. The methodology is termed Goal Oriented Commonality Analysis (GOCA). GOCA proposes two layers of modeling, including the theoretical model and the computational model, to resolve the conflict between the continuous mathematical models that represent the underlying theories of SC problems and the discrete nature of a computer. In addition, the theoretical model and computational model are developed to be abstract and documented

separately to improve reusability. Explicitly defined and documented terminology for models and requirements are included in GOCA, which helps avoid ambiguity, which is a potential source of reduced reliability. The traceability of current and future changes is used to potentially improve reusability and maintainability.

FASCS includes a Family Member Development Environment (FMDE) for the automatic generation of family members. FMDE is apparently the first complete environment that facilitates automatically generating variable code and test cases for SC program families. The variable code for a specific member of the program family can be automatically generated from a list of variabilities written in a Domain Specific Language (DSL), which is considerably easier than manually writing code for the family member. Some benchmark test cases for the program family can also be automatically generated.

Since both family members and test cases can be automatically generated, testing the program family can be performed on the same computational domain with different computational variabilities. This provides partially independent implementations for which test results can be compared to detect potential flaws. This capability partly addresses the unknown solution challenge for SC software.

Documentation is also an important part of FASCS. Five new templates for documenting requirements and design are proposed. Traceability matrices, which provide relations between artifacts (and documents) in the different stages of the process, can facilitate understanding of the programs. The matrices can also improve reusability and maintainability by helping trace

changes.

Nonfunctional requirements, especially nonfunctional variable requirements, are rarely considered in the development of program families. To the knowledge of the author, nonfunctional variable requirements have never been considered in the development of SC program families. Since some nonfunctional requirements are important for SC software, FASCS proposes using some decision making techniques, such as the Analytic Hierarchy Process, to rank nonfunctional variable requirements and select appropriate components to fulfill the requirements.

Acknowledgements

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

I am grateful to Dr. Qiao and Dr. Khedri for reviewing of this thesis and giving me valuable feedbacks and suggestions. Thanks to my colleagues for their help and friendship.

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

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

Hamilton, Ontario, Canada Wen Yu

January, 2011

Symbols, Acronyms and Abbreviates

Ω	computational domain
$\boldsymbol{\sigma}$	stress
σ	normal stress
τ	shear stress
$\boldsymbol{\epsilon}$	strain
ϵ	normal strain
γ	shear strain
ν	Poisson's ratio
ξ, η	local coordinates
$\mathbb{B}$	boolean type
$\mathbb{N}$	natural number type
$\mathbb{R}$	real type
$\mathbb{S}$	string type
$\mathbf{B}$	kinematic matrix
$\mathbf{D}$	constitutive matrix
E	Young's modulus
F	false
$\mathbf{F}$	constitutive load vector
$\mathbf{L}$	linear differential operator
$\mathbf{K}$	stiffness matrix
$\mathbf{N}$	shape function
S_T	boundary of traction

S_U	boundary of prescribed displacement
T	true
a	vector of displacement at nodes
b	body force
t	traction
u	general displacement
u, v, w	displacement in x, y and z direction
1D	one dimensional
2D	two dimensional
3D	three dimensional
AC	Anticipated Change
AD	Application Design
AI	Application Implementation
AHP	Analysis Hierarchy Process
AOP	Aspect-Oriented Programming
ARE	Application Requirement Engineering
AT	Application Testing
CBD	Component-Based Development
CCA	Common Computational Assumption
CM	Computational Model
CMS	Computational Model Specification
CR	Common Requirement
CTA	Common Theoretical Assumption

CTD	Computational Terminology Definition
CVRS	Common and Variable Requirements Specification
CVT	Computational Variability Test
DD	Domain Design
DI	Domain Implementation
DM	Direct Method, Domain Model
DRE	Domain Requirement Engineering
DSL	Domain Specific Language
DT	Domain Testing
FASCS	Family Approach for developing Scientific Computing Software
FCR	Functional Common Requirement
FDM	Finite Differential Method
FEM	Finite Element Method
FFEMP	Family of Finite Element Method Programs
FG	Functional Goal
FMA	Family Member Assembler
FMDE	Family Member Development Environment
FMGP	Family Member Generation Process
FVR	Functional Variable Requirement
GC	General Constraint
GOCA	Goal Oriented Commonality Analysis
GP	Generative Programming

M	Module
NCR	Nonfunctional Variable Requirement
NG	Nonfunctional Goal
NVR	Nonfunctional Variable Requirement
OO	Object Orientation
PDE	Partial Differential Equation
PMGT	Parallel Mesh Generation Toolbox
PSE	Problem Solving Environment
RC	Requiring Constraint
RMG	Reference Module Guide
RMIS	Reference Module Interface Specification
SC	Scientific Computing
SE	Software Engineering
SMIS	Specific Module Interface Specification
SPLE	Software Product Line Engineering
SVRS	Specific Variable Requirement Specification
TCG	Test Case Generation
TM	Theoretical Model
TMS	Theoretical Model Specification
TTD	Theoretical Terminology Definition
UC	Unlikely Change
VCG	Variable Code Generation
VR	Variable Requirement

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scientific Computing (SC) programs use models to simulate phenomena. These models are usually sets of continuous mathematical equations and solving these equations often requires large amounts of calculation. Since calculation is a key characteristic of this type of SC software, considerable time and effort have been invested in improving the quality factors that relate to calculation, such as accuracy, precision and efficiency. However, other quality factors, such as reusability, reliability, usability and maintainability, have not seen as much attention and still show room for improvement.

A proposed family driven methodology, FASCS, which is an acronym for Family Approach for developing Scientific Computing Software, is aimed at improving the overall quality of SC software. A program family is a set of programs that have some common features, called commonalities. Instead of developing a single program, a family of programs is developed. A program of interest is a member of the family. Each family member is differentiated

from the others by one or more variabilities. The major contribution of the proposed methodology, FASCS, is the adaption of the family approach to the development of SC software.

Since the quality of SC software is the motivation of the proposed methodology, the first section of this chapter is devoted to the definition of software quality (Section 1.1). The proposed methodology uses a family approach. Hence, the overall process of general program family development is given in Section 1.2. Finally, the organization of the thesis is summarized in Section 1.3.

1.1 Definitions of Software Quality

There is no standard definition of software quality. In this research, quality factors (Yu, 2007) are used to measure software quality. These factors are originally proposed by McCall et al. (1997). The quality factors in this research refer to the quality factors for a program. Henceforth, to avoid confusion, the word "program" refers to a general program, which may be a member of a program family that is developed using a family approach, or may be a program that is developed without using a family approach. Otherwise, the phrase "member of a program family" refers to a member of a program family that is developed using a family approach and the phrase "single program" refers to a program that is developed without using a family approach. The word "program" refers to a program as a whole, which includes code and documentation.

Eleven quality factors are defined in Yu (2007). However, only the definitions that relate to the current research are highlighted in this section.

Reusability: Extent to which a program can be used in other applications.

A member of a program family can be reused by other members of the same program family, by a member of other program families, or by a single program. The first case occurs most frequently, since members in the same program family solve similar problems.

Usability: Effort required for learning, operating, preparing input, and interpreting output of a program.

The assessment of the usability for a member of a program family is the same as that for a single program. For SC programs, preparing input and interpreting output usually requires more effort than actually operating the program.

Sometimes, reusability and usability are difficult to distinguish. If a program family is easy to migrate from one member to another member, one may say that the program family is easy to use and that the usability is improved. However, as mentioned, the definitions of quality factors in this research refer to a program, which can be a member of a family or a single program. If a program family is easy to migrate from one member to another, then a portion of one family member is easily reused by another family member. Hence, according to the above definitions, the reusability of a family member is improved.

A rule to identify whether a problem belongs to reusability or usability is to identify the subject of the problem. If the problem occurs for a developer of a program, then the problem belongs to reusability. On the other hand, if the problem occurs for a user of a program, then the problem belongs to usability.

Reliability: Extent to which a program can be expected to perform its intended function with the required accuracy and precision.

Reliability is important for an SC program. However, sometimes, reliability is difficult to judge. Some expected results for SC programs are unknown a priori, especially for programs that simulate phenomena that is "too complex, too large, too small, too dangerous, or too expensive to explore in the real world" (Segal and Morris, 2008). This is the unknown solution challenge for SC programs, which is also termed the oracle problem. The unknown solution challenge was discussed in Yu (2007). This challenge, together with other challenges for SC software, will be discussed in Section 2.2.1.

Improved reusability may increase reliability. Reliability problems may be due to errors in the program. When a program is reused, it is very likely that errors can be detected and fixed. Hence, reliability is improved. On the other hand, improved reliability may reduce reusability, since improving reliability may require adding more constraints to the program. These constraints may reduce the scope of the program and prevent other programs from reusing it.

Maintainability: Effort required for modifying an operational program.

The purpose for the modification includes to correct discovered problems, to keep the program usable in a changed or changing environment and to improve performance or other quality factors (Lientz et al., 1978). Expanding the scope of the program is also included in maintenance.

Maintainability strongly relates to reusability. The necessary condition for high reusability and maintainability is that the program be easy to change. To modify a program, the developer of a program must be able to easily locate the portions to be modified. This involves the understandability of a program. The easier a program is to understand, the easier the program is to modify. Hence, understandability can improve both of reusability and maintainability.

1.2 Overview of Program Families

Since the proposed methodology uses a family approach, this section will give a general introduction on program family and the process of using a family approach to develop software. A program family is a set of programs that share a common, managed set of features and that are developed from a common set of core assets in a prescribed way (Pohl et al., 2005). A program family approach is sometimes called Software Product Line Engineering (SPLE).

The family approach for developing software was conceived in 1970's (Parnas, 1976). The idea was inspired by the success of developing families of industrial products, such as cars. However, the program family approach did not get much attention from software developers until a decade ago (Clements and Northrop, 2002; SEI, Retrieved June 2010; Pohl et al., 2005; Weiss and

Lai, 1999).

The primary motivation for developing programs as a family is to produce customized products at reasonable costs (Pohl et al., 2005). The benefits of using a program family approach also include reducing the development cost and the time to market, and improving quality and productivity.

The basic assumptions that need to hold to make program family development worthwhile are (Weiss and Lai, 1999):

- The redevelopment hypothesis: Programs that can be developed as a family have some functions in common.
- The oracle hypothesis: The changes to the programs are predictable.
- The organizational hypothesis: The modification for one predicted change does not have a strong dependence on the modification of other predicted changes.

The program family approach introduced in this section provides background on the general approach. The specific methodology developed for this research is discussed in Chapter 3. The general program family approach discussed in this chapter is based on Clements and Northrop (2002), SEI (Retrieved June 2010), Pohl et al. (2005) and Weiss and Lai (1999). The approach presented here has been used by many program family practitioners (Czarnecki and Eisenecker, 2000; van der Linden et al., 2007; Voelter and Groher, 2007).

The program family approach has two basic phases: Domain Engineering and Application Engineering. Most researchers (Clements and Northrop, 2002; SEI, Retrieved June 2010; Pohl et al., 2005) explicitly divide each phase

into several stages, as shown in Figure 1.1. This same division is implicit in

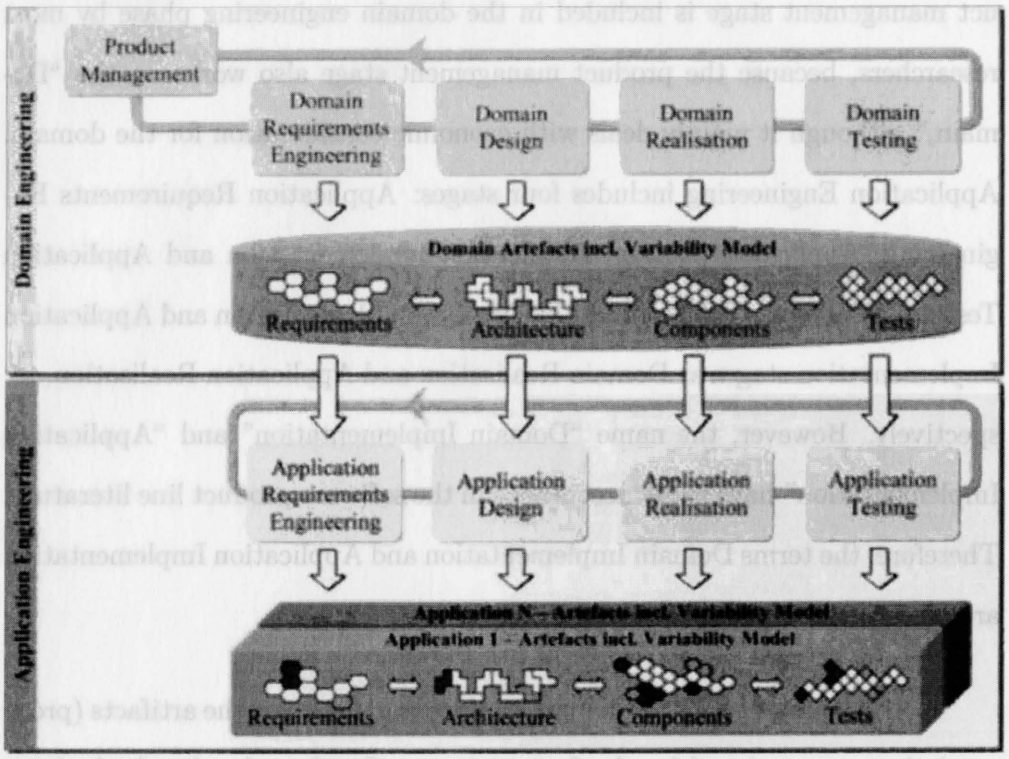

Figure 1.1: Overview of the Program Family Approach (Pohl et al., 2005)

Weiss and Lai (1999), where artifacts from the different domain engineering stages are combined into a development environment. The environment provides facilities for application engineers to develop family members. The use of the environment is adapted by the proposed methodology, FASCS, and will be introduced in Chapter 3 when FASCS is specifically discussed.

The Domain Engineering phase includes five stages: Product Management, Domain Requirements Engineering, Domain Design, Domain Implementation and Domain Testing. Weiss and Lai (1999) separate the product

management stage from the domain engineering phase. However, the product management stage is included in the domain engineering phase by most researchers, because the product management stage also works on the "Domain," although it mainly deals with economic consideration for the domain. Application Engineering includes four stages: Application Requirements Engineering, Application Design, Application Implementation and Application Testing. Pohl et al. (2005) names the Domain Implementation and Application Implementation stages as Domain Realisation and Application Realisation, respectively. However, the name "Domain Implementation" and "Application Implementation" have wider acceptance in the software product line literature. Therefore, the terms Domain Implementation and Application Implementation are used in this thesis.

The later stages in the domain engineering phase use the artifacts (products) that are produced by the former stages. On the other hand, the later stages in the application engineering phase use both the artifacts that are produced by the former stages in the application engineering phase and the artifacts that are produced by corresponding stages in the domain engineering phase. For example, the application design stage uses the Requirements produced by application requirements engineering and the Reference Architecture produced by the domain design. As for developing a single program, the processes of both domain engineering and application engineering are iterative. The later stages provide feedbacks for former stages in both the domain engineering phase and the application engineering phase. In addition, the stages in the application engineering phase also provide feedback for the corresponding

stages in the domain engineering phase. The process for developing program family is outlined in the rest of this section.

1.2.1 Domain Engineering

During the domain engineering phase, the economics of the program family are analyzed, the common and variable requirements of the program family are defined and the commonalities are designed, implemented and tested. The product of domain engineering is the "platform," on which all family members are based.

1.2.1.1 Product Management

Product management is the first stage of the whole process. During this stage, the economics of the program family are analyzed and the scope of the program family is determined. The output of this stage is the major common and variable features of the program family and the schedule for development.

1.2.1.2 Domain Requirements Engineering

Domain requirements engineering is sometimes called commonality analysis. It is a very important stage for the development of a program family because all common and variable requirements are developed in this stage. These common and variable requirements are the basis for the later development. The output of the commonality analysis stage is the common and variable requirements for the program family.

No one software requirements eliciting process is the best for all types

of software, nor is there a single best format for documenting software requirements. Similarly, the process for developing the common and variable requirements and the way that these requirements are documented can be different for different domain areas. For example, the process used for the commonality analysis of a Floating Weather Station, which is an example in Weiss and Lai (1999), and for Home Automation, which is an example in Pohl et al. (2005), are different. Moreover, the formats for documenting common and variable requirements for the two examples are also different. The common and variable requirements for the former example are documented textually and those for the later example are documented graphically.

1.2.1.3 Domain Design

During the domain design stage, a reference architecture for the program family and a model that specifies differences for some family members is developed. A variety of names for the model are used in the literature. However, the essence is the same. The name Domain Model is used in this section.

A reference architecture provides the structure for all members of the program family. Hence, it should be flexible enough to accommodate all variable requirements. At the same time, it should also contain enough information for application engineers to develop individual family members. The reference architecture can be designed using module decomposition (Parnas, 1972). However, this approach is not mandatory.

A language that specifies variable requirements, which is usually a Domain Specific Language (DSL), is the key part of the domain model. Since

family members are distinguished by variable requirements, a family member can be specified by a program written in this language.

To improve the usability of the program family, automatic generation of program family members has become prevalent. As discussed in Section 1.1, improved usability of the program family can improve the reusability of the family members. There are two aspects that can be automated: automatic generation of code (Carette, 2006) and automatic assembling of code (Czarnecki and Eisenecker, 2000). Both the automatic code generator and the automatic assembler are designed in this stage, if the automatic approaches are to be used.

Another way to improve the usability of the program family is to reuse existing components. These components can be pieces of code or binaries. Component-Based Software Engineering (Bachmann et al., 2000; Heineman and Councill, 2001) focuses on how to make the components exchangeable so that they can be reused.

1.2.1.4 Domain Implementation and Testing

In the domain implementation stage, code that is common to all family members is developed. Developing common code may include reusing existing components. If code is automatically generated, the code generator, which is designed in the domain design stage, is implemented and common code is generated. If the automatic configuration approach is used, a component assembler, which is also designed in the domain design stage, is implemented.

The domain testing stage deals with testing code that is common to

all family members and testing the code generator, if the code is to be generated automatically. The configuration mechanism is tested if the automatic configuration approach is used to generate family members.

1.2.2 Application Engineering

The members of a program family are developed in the application engineering phase. The process for developing family members is much simpler than the process for developing single programs, since artifacts produced in the domain engineering phase can be reused. The application engineering phase cannot start until the domain engineering phase starts. However, it is not necessary that the application engineering phase waits for the domain engineering phase to finish. The activities for domain engineering and for application engineering can work in parallel. The stages in the application engineering phase can start after the corresponding stages in the domain engineering phase finish. In fact, it is often the case that one or more members of the program family are developed in parallel with the development of the program family. A benefit of parallel development is the reduction of the time to market of family members. In addition, the activities in the application stages provide the benefits that they give feedback to the corresponding stages in domain engineering for refinement of the domain engineering artifacts. This improves the quality of the program family.

The application engineering starts with the application requirements engineering stage. The output of this stage is the application requirements specification. The common requirements are already specified in the com-

monality analysis stage and can be reused. The variable requirements for the specific family member are now determined.

The application design stage uses the artifacts from domain design stage and application requirements engineering stage. Depending on the values of variable requirements for the family member, the reference architecture is configured to the family member to be developed. The output of this stage is the architecture for the specific family member.

In the application implementation stage, only variable code is developed. Common code that was developed in the domain implementation stage is reused. The variable code is developed using a traditional approach or an automatic code generation approach. The common code and variable code are also configured, manually or automatically, in this stage

During the application testing stage, tests of the specific family member are performed. These tests include the unit test for variable code. The integration test and system test are the same as for testing a single program.

1.3 Outline of Thesis

This chapter (Chapter 1) gives background information on software quality and program families. Chapter 2 specifies the quality concerns for SC software, which are the motivations of the proposed methodology. The proposed methodology, FASCS, is introduced in Chapter 3, followed by the detailed discussions on the methodology. Chapter 4 presents GOCA, a new methodology of commonality analysis for FASCS. Chapter 5 specifies another high-

light for the proposed methodology, the Family Member Development Environment. Chapter 6 specifies how implementation and testing are performed using FASCS, where a new test technique called Computational Variable Test is proposed. Chapter 7 summarizes the documents for developing SC program families using FASCS, where five new templates for documenting the development of SC program families are proposed. The discussion of the documents (Chapter 7) is separated from the development stages (Chapters 3 – 6) to emphasize the importance of the documentation and not to disrupt the flow of the presentation for the stages. Finally, the conclusion and future works are provided in Chapter 8.

Chapter 2

Quality Concerns for Scientific Computing Software

The previous chapter (Chapter 1) gave the background information on software quality and the program family concept for general software. This chapter will focus the discussion on Scientific Computing (SC) software. First, some quality problems for SC software are introduced. Then, causes for unsatisfactory SC software quality are discussed, followed by a summary of previous endeavors of researchers to overcome these quality problems. Finally, an example SC program family, FFEMP, is introduced.

2.1 Quality Problems for Scientific Computing Software

SC software has been successfully used in a variety of applications. For instance, it is 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 (Yu and Smith, 2009).

SC software can be commercial or noncommercial. This research focuses on noncommercial SC software, in particular, on noncommercial SC software that is developed by professional end user developers (Segal and Morris, 2008), such as scientists or engineers. SC software that is developed by professional end users is quite prevalent in research institutes and universities.

Many of the end user developed SC programs have very similar functionalities. The proliferation of programs with similar functionality can be seen by considering the long lists of similar SC programs available on the Internet. An example is the list of FEM programs on Young and MacPhedran (Retrieved January 2011), which has over 120 public domain finite element programs. The proliferation of similar programs suggests that reusability is a quality factor of SC software that can be improved.

Reliability is also a concern for SC software. Some failures of applications are caused by inaccurate computations from embedded SC software. The failure of an American Patriot Missile battery in Dharan, Saudi Arabia, to intercept an incoming Iraqi Scud missile in 1991 (Cirincione, 1992) is one of many examples. Failures also occur in SC programs used to check theo-

ries developed by scientists. For example, the failure of the analysis program used by Geoffrey Chang to discover the structure of a protein called MsbA resulted in the retractions of five of his papers (Miller, 2006). These examples demonstrate that there is room for improvement in reliability for SC software.

Many end user developed SC programs are used by professors in universities for checking their scientific theories and the developers of these programs are usually their graduate students. This fits with the findings that the lifetime of SC programs used in universities is usually less than 5 years (Tang, 2008), since this is the duration of a typical PhD program. On the other hand, developing a scientific theory often requires scientists working for many years. This means that the lifetime of the program is shorter than the time for developing a theory, which indicates that the use of the program is discontinued. One potential reason for the short lifetime of some end user SC programs is the usability challenge. That is, the programs are difficult to use, so researchers cannot use the programs written by other researchers. Another potential reason is the maintainability challenge. Scientific theories under development may constantly evolve. Hence, programs for testing the theories also need to evolve. However, using current development approaches, the evolution may be more difficult to maintain than developing a new program. Therefore, researchers usually develop new programs instead of spending time on modifying existing ones.

The above quality problems are considered by the proposed methodology, FASCS. FASCS uses a program family approach, which can improve reusability, usability, reliability and maintainability, as discussed in Section

1.2. In addition to the process, FASCS also uses many techniques to further improve quality. Some techniques to improve SC software quality and the adoption of these techniques by FASCS will be introduced later in this chapter (Section 2.3.4).

2.2 Potential Reasons for Quality Problems in Scientific Computing Software

The potential reasons for the quality problems mentioned in Section 2.1 fall into two categories: characteristics of SC problems, and the ways that SC software is developed.

2.2.1 Characteristics of Scientific Computing Software

Some characteristics of SC software can cause challenges for its development. Some of these challenges, such as the approximation challenge and the unknown solution challenge, are unique to SC software. Others are not unique, but these challenges occur very frequently in the development of SC software. The challenges for developing SC software are listed below. How the proposed methodology, FASCS, address these challenges will be presented when the details of FASCS are discussed in the rest of this thesis.

Approximation Challenge Most real numbers are approximated by floating point numbers on a computer. This approximation is one of the causes for reliability issues. The failure of an American Patriot Missile,

mentioned in Section 2.1, was caused by the round off errors of floating point numbers. In addition, truncation of continuous or infinite numerical algorithms to discrete and finite approximations is also a source of errors, called truncation errors. More discussions on analysis of round off errors and truncation errors can be found in Heath (2003).

Unknown Solution Challenge The unknown solution challenge (Smith and Yu, 2009) is another characteristic that impacts reliability. Many SC programs are used to solve problems whose true solutions are unknown. That is, there are limited test cases with known solution for the programs, which means an incomplete test oracle (Sanders and Kelly, 2008). Hence, it is difficult to assure the reliability of the programs because the accuracy and precision are difficult to judge without a proper test oracle. This difficulty is different than other types of software, such as embedded and real-time systems, whose true solutions are known although these solutions are difficult to obtain.

Technique Selection Challenge Real world problems that interest scientists and engineers are modeled before they can be solved. These models are usually represented by continuous mathematical equations. However, these continuous equations cannot be directly solved by a computer due to the discrete nature of a computer. Some techniques (algorithms) are available to numerically solve continuous equations. For example, there are many techniques, such as the Finite Element Method, for solving Partial Differential Equations, which model many physical problems. These

techniques are usually different from one another by some quality factor, such as reusability and reliability. That is, the selection of the appropriate technique depends on the nonfunctional requirements of the SC software. However, nonfunctional requirements are difficult to specify and assess. How to select the appropriate technique for solving these continuous equations is a challenge that is typically left as a decision for the domain expert, but ideally there would be a systematic means to make the decision related to nonfunctional requirements.

Input Output Challenge The equations that model the problems of interest usually require considerable amounts of input data and produce large volumes of output. It is not uncommon to have output files with sizes measured in megabytes, or even gigabytes. The interpretation of this vast amount of data for the input and output is often complicated. As Dubois (2002) mentioned, people often "reinvent the wheels" even though library routines for solving their problems exist because of the difficulty of preparing input data and interpreting the output results. The characteristic of complicated input data and output results creates challenges for the usability of SC software.

Modification Challenge SC software that is used by some scientists needs to be flexible to accommodate potential changes in their research. The change of requirements also occurs in other types of software. However, the frequency of the change is sometimes very high for SC software. This is one of the reasons that some scientists do not use commercial software,

since it is usually difficult or expensive to quickly modify existing commercial software according to the users' requirements. Unfortunately, the noncommercial programs are often not as flexible to change as would be expected, as the developers do not usually develop their programs for reuse (Segal, 2008).

2.2.2 Patterns of Scientific Computing Software Development

In addition to the characteristics of SC software, the development patterns of SC software can also cause quality concerns. Software engineering methodologies have not been widely used for developing SC software, especially the SC software that is developed by professional end users (Segal, 2008; Wilson, 2006). One of the reasons is that the professional end user developers, scientists and engineers, have extensive domain knowledge, but little background in software engineering. Only 13% of SC software developers in recent survey identified themselves as having some software engineering education in their background (Tang, 2008).

In fact, many end user SC programs consist of nothing other than code. A common development model is to start a new project by copying the code from a similar project and modifying it for the new use (Yu and Smith, 2009). The copy and modify development approach results in significant maintenance headaches because of the challenge of propagating future improvements back to the ancestors of a given program. Reusability also suffers because, although much of the code is reused, the reuse occurs in an ad hoc manner, with success

depending on the programmer having intimate knowledge of the implementation details of their code.

In addition, the frequent lack of systematic testing causes challenges for reliability. Testing is usually not thorough for professional end user developed software. The lack of testing data, which is mentioned in Section 2.2.1, is one of the reasons for this. Another reason is that end user developers usually do not have a clear idea of how to test software. Most SC programs are only tested using test cases with analytical solutions for the equations that model the problems. An example of incomplete and redundant test cases is the prototype software system used to estimate radiation doses from cosmic sources in the Department of Chemistry and Chemical Engineering at the Royal Military College of Canada (Kelly et al., 2010).

The unprofessional development of SC software brings an understandability problem, which brings associated reusability problems. Although the programs developed by end users are often open source, these programs are usually not well documented and their understandability is a challenge. Hence, reuse does not occur as often as would be desirable.

In addition, there is a trust issue in the development of SC software. The end user developers do not often consider the reuse of programs, except in the case of some libraries. The reason is that they do not "trust" the existing programs. They usually want to understand the code before they can reuse it. This is part of the reason why they refuse to use some commercial software because they cannot obtain and investigate the source code.

The above SC software development patterns are used by end user de-

velopers for years and have their own benefits, such as short development time. It is not feasible for these end users to thoroughly learn Software Engineering (SE) knowledge before they develop software. A methodology for developing SC software should not require an extensive SE background and should be easy to follow.

The proposed methodology, FASCS, is easy to follow. It uses the family approach introduced in Section 1.2. If it is completely followed, end user developers, who act as application engineers, will not have to write any code, except a very simple Domain Specific Language (DSL) program. This addresses the problems caused by the development methodologies mentioned above and has the benefit of short development time for the end user developers.

To judge the quality of FASCS, the systematic approach proposed by Carver et al. (2007) for studying SC software could be pursued. The advantages of using FASCS can be shown by adapting FASCS to a real SC project and comparing the interesting quality factors of a program developed using FASCS with the same quality factors of a program developed using the existing method. The feedback from the development of the project is also valuable for improving FASCS.

2.3 Methodologies for Improving the Quality of Scientific Computing Software

SC is one of the primary motivations for inventing computers. However, improvement to the development of SC software have not seen much attention.

Although some of the problems have been reported (Carver et al., 2007; Kelly, 2007; Segal and Morris, 2008), not many methodologies have been specifically developed for SC software to address these problems. The methodologies that have been adopted to address SC quality concerns are summarized in this section.

2.3.1 Object-Orientation

Object-Orientation (OO) (Meyer, 1988) is a software development approach that has been successfully applied to developing programs, such as business applications, to improve their quality. The OO approach improves software quality through encapsulation, inheritance and polymorphism. However, there is an understandability challenge brought by these three OO features, although the advantages of the OO approach are generally considered to outweigh this challenge.

By using encapsulation, information that is unnecessary for other classes is hidden inside one class. This is the basic software design principle, information hiding (Parnas, 1972), that has been widely accepted by software engineering community. Many SC programs are also developed using this principle.

Inheritance may bring problems for developing SC software. The reason for the success of applying the OO approach to some domains is that the Object is the key to the application and inheritance naturally exists among the objects. For example, an object, Account, is one of the most important aspects for an automatic banking machine (ATM) system. A bank account

may be a Savings Account or a Chequing Account. However, both Savings Accounts and Chequing Accounts have many features in common. Hence, one can create an Account class as a parent class and make Savings Account and Chequing Account inherit the Account class and become its child classes.

Unlike the above example, the central part of many SC problems is algorithms (Berti, 2000), not objects, and algorithms are "flat" (Di Felice, 1993). That is, inheritance does not naturally exist. For example, it is difficult to find the inherited relationship between Gaussian Elimination and Gauss Seidel, which are two algorithms to solve linear systems of equations. Although these two algorithms can be designed as child classes of an algorithm class, there is little in common since these classes usually do not have any class field. That is, inheritance would be an unnatural or forced concept that would make the programs more difficult to understand compared with a program that does not use inheritance.

It is possible to declare the above algorithm class as an abstract class and let Gaussian Elimination and Gauss Seidel be its subclasses. Then, the algorithm to be used is dynamically bound at runtime. This polymorphism can improve reusability. However, the dynamic binding may potentially delay the time of discovering errors, since the type checking cannot be done at compile time. Since the algorithm for solving a specific problem using SC software is usually determined when the program is implemented, the binding time can be moved to compile time. That is, instead of a general program that can solve a problem using different algorithms, a family of programs using different algorithms is developed. This is what a program family approach

suggests. The program family approach will be further discussed in Section 2.3.3.

Moreover, the objects in SC software are data. Although some researchers try to separate data structures from algorithms (Berti, 2000; ElSheikh et al., 2004), most data structures in SC software relate to algorithms. The data structure may be different depending on algorithms to achieve optimal accuracy, precision or efficiency. Accommodating these differences into one parent class may make the data structure difficult to understand. For example, the understandability challenge exists in many Finite Element Method (FEM) programs that are developed using OO approach, such as FEMOOP (Martha, 2002), OOFEM (Patzák, 2000; Patzák and Bittnar, 2001a) and OFELI (Touzani, 2002). To understand a class that is meaningful to a FEM model, such as a 2D beam element, in one of above FEM programs, OOFEM, one must understand its 7 ancestor classes. On the other hand, a 2D beam element in a program without inheritance, such as the program in Stolle (2008), is represented by a list of nodes, which are represented by their coordinates. It is clear that the program in Stolle (2008) is easier to understand than OOFEM.

Nevertheless, the encapsulation of OO is still an excellent way to hide information. OO design and OO programming can be used for developing SC software, but with careful use of inheritance.

2.3.2 Agile Methods

Another software development approach, agile methods, has become more and more prevalent in the software engineering community. This method-

ology copes well with software with unpredictably changing requirements. It has been successfully applied to some SC software projects (Easterbrook and Johns, 2009; Kane et al., 2006; Wood and Kleb, 2003). However, there are limitations for applying agile methods. For example, agile methods do not benefit projects having "relatively well-defined and stable requirements" (Crabtree et al., 2009). There is also not much benefit for using agile methods if the requirements change predictably. Programs with predictably changing requirements, which are common in SC, can be developed as a program family, which will be discussed in the next section (Section 2.3.3).

2.3.3 Program Family Approach

In contrast with agile methods, a program family approach is a traditional "plan-driven" approach, meaning that it is suitable for projects with relatively stable or predictably changing requirements.

The program family approach improves reusability because, under this approach, "reuse is planned, enabled, and enforced" (Clements and Northrop, 2002). Since the common portion of the program family is reused and retested, defects are more likely to be discovered than if the program is tested only once. Hence, reliability is improved. Each member of the program family is not a general purpose program. It solves a small set of specific problems. A general purpose program is usually more difficult to use than a specific one because some conditions have to be set to "tell" the general purpose program the features of the problem to be solved. For example, if a general purpose program for multiplying matrices is used to multiply two sparse matrices,

we must "tell" the program about the sparseness so that a more efficient algorithm can be used. When the problem to be solved becomes complicated, the setting of these kinds of conditions can also be complicated. Therefore, a specific program family member has better usability than a general purpose program. Maintainability is also improved because, unlike the copy and modify development approach mentioned in Section 2.2.2, the core assets for all family members are managed together.

The success of the program family approach in developing software promotes adopting this methodology to SC software development (Smith and Chen, 2004; Carette, 2006; Bastarrica and Hitschfeld-Kahler, 2006; Yu and Smith, 2009; Smith et al., 2010; Carette et al., 2011). However, these previous attempts only focus on some aspects of the program family approach. Smith and Chen (2004), Smith et al. (2010) and Yu and Smith (2009) emphasize commonality analysis. On the other hand, Bastarrica and Hitschfeld-Kahler (2006) focus on design and implementation and do not mention a commonality analysis. Carette (2006) and Carette et al. (2011) focus on automatic code generation. All stages of the SC program family development need to be explored to achieve completeness of the methodology, which is the primary purpose of this thesis.

Developing software as a family is a general approach. The success of applying it to a specific domain depends on it being tailored for that domain. The reason that SC programs can be developed as families is that many SC programs meet the three development hypotheses mentioned in Section 1.2.

- The redevelopment hypothesis: An SC program uses a model to sim-

ulate a phenomenon. There must be some assumptions that are true for the model to represent the phenomenon. It is very likely that the model with one or more modified assumptions can be used to represent a slightly different phenomenon. In this case, the program with some modification can be reused. For example, many programs using the Finite Element Method to solve elasticity problem in solid mechanics have the functionality of solving displacements. This indicates that there exists some SC programs that have some functions in common. That is, the redevelopment hypothesis for many SC programs holds.

- The oracle hypothesis: Many SC programs, especially professional end user developed SC programs, are used to solve equations that model the scientific theories that have been developed for many years, such as the governing equations for the elasticity problem in solid mechanics as Equation 2.1 will show in Section 2.4. Since the underlying theories are stable, the changes of this kind of programs are predictable. For an SC program that models a developing scientific theory, changes related to the computational decision, such as the potential range of the number of straight line segments to use to approximate a curve, are generally known in advance. In addition, other changes can usually be estimated by the scientists developing the theories. That is, the oracle hypothesis for many SC programs holds.

- The organizational hypothesis: As mentioned above, the change of assumptions for the model is the major source of modifications for SC pro-

grams. Since the changes of assumptions for the models are usually not strongly dependent, the modifications are usually not strongly dependent. A challenge related to organization hypothesis for SC programs, the connection between data structures and algorithms, was observed (Chen, 2003). However, some techniques have been proposed to separate the connection (Berti, 2000; ElSheikh et al., 2004). If it is necessary, the separation of data structures and algorithms can be achieved. The above discussion indicates that the organizational hypothesis for many SC programs holds.

The proposed methodology, FASCS, focuses on the breadth of the family approach to developing SC program families. This is the first time in the SC software development community, where all stages in both the domain engineering phase and the application engineering phase are included. In addition, FASCS also deeply explores some stages of the process, such as the commonality analysis. Unlike Carette (2006), which automatically generates all common and variable code, FASCS suggests automatically generating variable code and developing common code using a traditional approach. This combination of automatic and traditional approaches allows end user developers without knowledge of automatic code generation to modify the whole program family.

2.3.4 Some Techniques for Improving SC Software Quality

There are many techniques to improve software quality that can be used when developing SC software. The techniques discussed in this section do not contribute to an entire software development process. For example, none of these technique include a process for eliciting software requirements. However, these techniques can be used in the software development processes mentioned previously. For example, libraries can be used when the OO approach is used for software development. Many of these techniques are also adopted to FASCS. The adoptions are discussed when the techniques themselves are introduced.

2.3.4.1 Libraries

Using libraries is one of the traditional methods to improve reusability. Some of the libraries, such as GSL (GSL, Retrieved December 2010), BLAS (BLAS, Retrieved December 2010), LAPACK (Anderson et al., 22 Aug 1999) and NAG (NAG, Retrieved December 2010), have been used for many years. However, the effectiveness of using these libraries heavily depends on the experiences of the users of the libraries. As argued in Dubois (2002), "their market penetration is far below what it could be." The major reason is that users find "the large number and variety of arguments to be too intimidating" and users are confused when they need to set values for some of the arguments.

Libraries play an important role in SC software. Libraries can also be used in FASCS. The problem of setting arguments can be solved when using FASCS, because the end user developers act as application engineers.

They only need to write a program using a Domain Specific Language (DSL) to specify the member of a family they desired, which should be relatively simple. More details on the development of a DSL for SC program family will be presented in Chapter 5.

2.3.4.2 Component-Based Development

Component-Based Development (CBD) (Heineman and Councill, 2001) shares the same idea as libraries in the sense that they all focus on reusing units of a program. One of the differences is that a "component" is not restricted to a subroutine, as for a library. It can be a module, a package or even a binary. For a large distributed multiple business domain, the components are further abstracted into services, which can be passed through a web-based communication environment. This is an extension of CBD and is called service-oriented development (Papazoglou and Heuvel, 2007).

Another difference is that CBD is more thorough than the library approach because it also suggests some techniques for developing reusable components. One of the techniques is generic programming, which will be discussed next in Section 2.3.4.4. Moreover, CBD includes a process for configuring the components.

CBD has been used in SC software development, such as MPQC and NWChem, which are two quantum chemistry simulation packages (Alexeev et al., 2005). CBD can be seen as a portion of a program family approach, such as FASCS. In particular, developing components is included in the domain implementation stage and configuration is included in the domain design stage.

2.3.4.3 Aspect-Oriented Programming

Similar to the OO approach introduced in Section 2.3.1, aspect-oriented programming (AOP) (Kiczales et al., 1997) also focuses on the separation of concerns. However, AOP aims on aspects that cross-cut the system's basic functionality. It complements the OO approach by providing another way of thinking about program structure. The key unit of modularity in OO is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns that cut across multiple types and objects.

AOP has been used in the development of SC software. Irwin et al. (1997) developed sparse matrix code using AOP. Harbulot and Gurd (2004) demonstrated the possibility of using aspects for decoupling the implementation of parallelization from the implementation of the numerical models.

AOP isolates secondary or supporting functions from the main program's business logic. In this way, it is similar to the family approach (Section 2.3.3), which isolates variabilities from commonalities.

2.3.4.4 Generic Programming

Generic Programming is a programming paradigm for developing efficient, reusable software. The Standard Template Library (Plauger et al., 2000), which became part of the ANSI/ISO C++ standard, was the first major success of applying this paradigm. Generic Programming achieves reusability by finding commonality among similar implementations of the same algorithm, then providing suitable abstractions so that a single, generic algorithm can cover many concrete implementations (Indiana University, 2010).

Generic programming has been applied to develop SC software. Examples of using generic programming to develop SC software include a library supporting mesh-level geometry components (Berti, 2006), high-performance parallel code for solving two archetypal PDEs (Lee and Lumsdaine, 2002) and a high-performance vector mathematics library, Blitz++ (Blitz, Retrieved December 2001). Generic programming can be used in FASCS. As mentioned when CBD was discussed in Section 2.3.4.2, generic programming can be used to write reusable components. More details on how the generic programming fits within FASCS will be discussed in Chapter 5.

2.3.4.5 Generative Programming

Generative Programming (GP) is "a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge" (Czarnecki and Eisenecker, 2000). Compared to CBD discussed in Section 2.3.4.2, GP emphasizes the configuration so that the products can be automatically manufactured.

An example of using GP to develop SC software is the Generative Matrix Computation Library, given in Czarnecki and Eisenecker (2000). Arora et al. (2009) also discusses applying GP to developing SC software and gives a case study of developing a Poisson Solver, which is used for solving separable Partial Differential Equations. Carette et al. (2011) use a generative approach

to develop a program family of geometric kernels for mesh generation. As mentioned in Section 1.2.1.3, GP can be used in developing program families and FASCS uses a generative approach. Further detail on this topic will be given in Chapter 5.

2.3.4.6 Problem Solving Environment

A Problem Solving Environment (PSE) is a computer system that provides all the computational facilities needed to solve a target class of problems (Gallopoulos et al., 1994). A PSE provides advanced solution methods, automatic or semi-automatic selection of solution methods, and ways to easily incorporate novel solution methods. The use of libraries (Section 2.3.4.1) and PSEs can be linked together, so that the difficulties of using libraries can be solved (Rice and Boisvert, 1996).

Developing SC software is the major application of PSEs. Matlab (Mathwork, Last Access 2010) and Maple (Maplesoft, Retrieved 2010) are two successful PSEs developed to solve SC problems. The connection between SC and PSEs is summarized in Houstis et al. (1997). The idea of a PSE is used in the program family approach. In fact, the environment, which is an artifact of domain engineering, as introduced in Section 1.2.1, is a PSE. PSEs are also used in the proposed methodology, FASCS. The Family Member Development Environment (FMDE), which is an important part of FASCS, is a PSE. More details on FMDE can be found in Chapter 5.

2.3.4.7 Design Patterns

A design pattern is a general reusable solution to a commonly occurring problem in software design. It "describes a problem which occurs over and over again in our environment" (Alexander et al., 1977). A design pattern is a description or template for how to solve a problem that can be used in many different situations. Design patterns gained popularity after the book "Design Patterns: Elements of Reusable Object-Oriented Software" (Gamma et al., 1995) was published. Hence, design patterns are mainly used in OO design.

Design patterns are used in the development of SC software when the OO approach is applied. For example, Decyk and Gardner (2008) discusses the concept, application, and usefulness of software design patterns in Fortran95 based scientific programming. Blilie (2002) explores the application of patterns to dynamic-systems simulation, such as molecular dynamics, and identifies four design patterns that emerge in modeling such systems. Design patterns can be used for developing program families. One of the examples is the Facade design pattern. The interfaces for the variable portion of a program family is unified, although the internal designs are different.

2.4 An Example Scientific Computing Program Family: FFEMP

An example SC program family, FFEMP, is used to illustrate how to use the proposed methodology, FASCS, to develop an SC program family when FASCS is discussed in the rest of this thesis. This example program family is briefly

introduced in this section.

FFEMP stands for Family of Finite Element Method Programs. It is a proof of concept program family and can solve elasticity problems in solid mechanics. Any member of the FFEMP can solve for the displacements of nodes and the displacements of any point in the computational domain. Some members can solve for and output the stress and/or strain. The displacement, together with the stress and strain, are computed by solving the following set of Partial Differential Equations (PDEs):

$$\nabla\boldsymbol{\sigma} = 0 \tag{2.1a}$$

$$\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\epsilon} \tag{2.1b}$$

$$\boldsymbol{\epsilon} = \mathbf{L}\mathbf{u} \tag{2.1c}$$

$$\mathbf{t}^{(\hat{\mathbf{n}})} = \bar{\mathbf{t}} \text{ on } S_T \tag{2.1d}$$

$$\mathbf{u} = \bar{\mathbf{u}} \text{ on } S_U \tag{2.1e}$$

$$S_T \cap S_U = \emptyset \tag{2.1f}$$

Figure 2.1 shows an example problem that can be solved by a member of FFEMP. The computational domain for the example is rectangular.

In Equation 2.1, $\boldsymbol{\sigma}$ represents stress, $\boldsymbol{\epsilon}$ represents strain and $\mathbf{u}$ represents displacement. Equation 2.1a is called the Equilibrium Equation. It shows that the sum of all forces and moments is zero. Equation 2.1b is called the Constitutive Equation. It shows the relation between the stress and the strain, where $\mathbf{D}$ is the Constitutive Matrix. Equation 2.1c is called the Kinematic

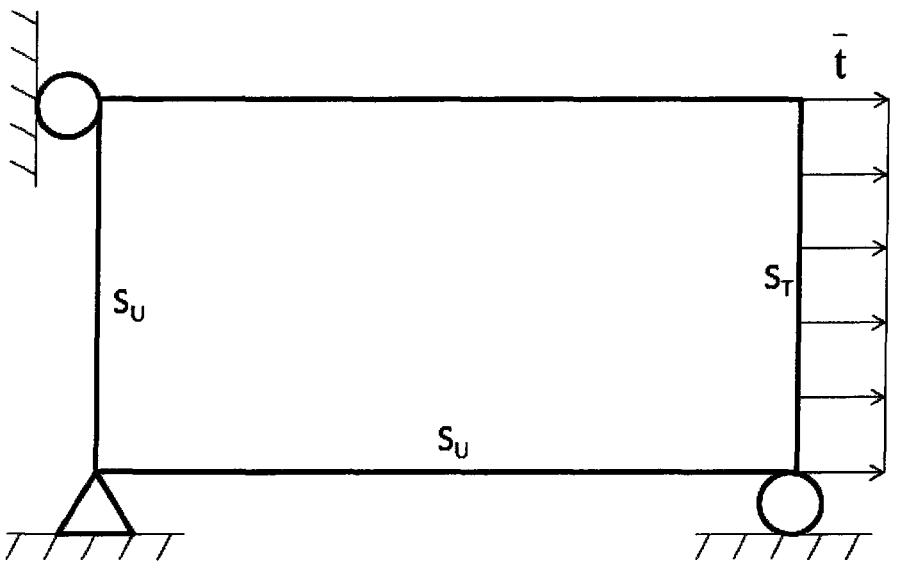

Figure 2.1: An Example Elasticity Problem in Solid Mechanics

Equation. It shows the relation between the strain and the displacement, where **L** is a Linear Differential Operator. Equation 2.1d gives boundary conditions on traction, where S_T represents the boundary for the tractions. In the example, it is the right side of the rectangle. The tractions on the right side is $\bar{\mathbf{t}}$ and they are equally distributed. Equation 2.1e gives boundary conditions on prescribed displacement, where S_U represents the boundary for the prescribed displacement. In the example, it is the left and lower side of the rectangle. The prescribed displacement $\bar{\mathbf{u}}$ on the left side are zeros in the horizontal direction and on the lower side is zero in the vertical direction. More details on the equation, such as the assumptions, input and output and the formal definitions of the symbols, can be found in the Theoretical Model Specification (Yu, 2010b). This equation will also be discussed in Chapter 4.

FFEMP solves for a typical SC problem, which can be modeled as a continuous mathematical equation, as described in the beginning of this chapter. The model that FFEMP solves is Equation 2.1, which is a set of PDEs. When the domain of interest becomes complicated, the input data becomes very large and the amount of calculation for solving problems becomes significant. In these cases, it is difficult, sometimes even impossible, to solve the problems without the help of a computer.

As for typical SC software, FFEMP solve the problems approximately. It uses the Finite Element Method (FEM), which is a numerical technique to solve PDEs. To use FEM, the computational domain must be decomposed into a set of small and simple shapes called elements. One possible decomposition of the computational domain shown in Figure 2.1 is given in Figure 2.2. The

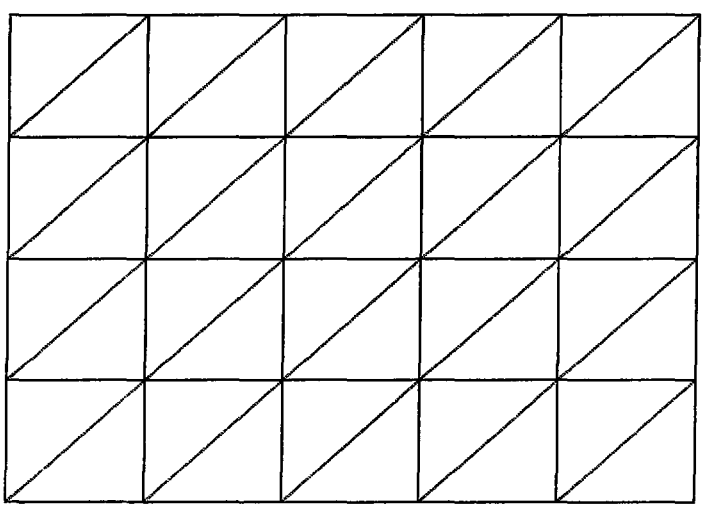

Figure 2.2: An Example Mesh

shapes of the elements in the example mesh are triangular. An example of

a single element is shown in Figure 2.3. In this example, each element has 3

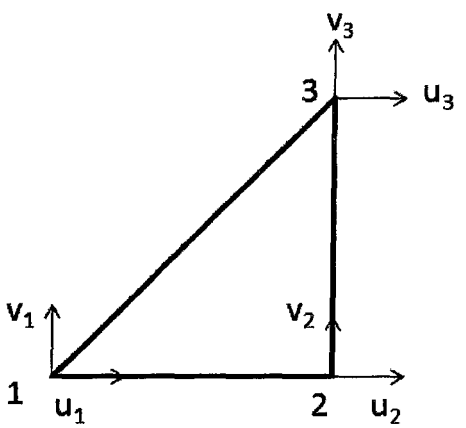

Figure 2.3: An Example Triangular Element

nodes. Each node has 2 degrees of freedom for the displacements: u_i and v_i, where $i = 1, 2, 3$. By using FEM, Equation 2.1 becomes:

$$\mathbf{F} = \mathbf{Ka} \tag{2.2}$$

where $\mathbf{F}$ is termed the consistent load vector, $\mathbf{K}$ is the stiffness matrix and $\mathbf{a}$ is the vector of the displacements of the nodes. Equation 2.2 is the ultimate equation that FFEMP solves. Details on this equation and how it is derived can be found in the Computational Model Specification (Yu, 2010c). This equation will also be discussed in Chapter 4.

Chapter 3

An Overview of FASCS

This chapter outlines the proposed methodology: FASCS, which is an acronym for Family Approach for developing Scientific Computing Software. FASCS suggests developing scientific computing programs as a program family. As discussed in Section 2.3.3, a program family approach can improve the reusability, reliability, usability and maintainability of SC software. This approach also addresses the modification challenge that is introduced in Section 2.2.1, as Section 3.1.2.2 will discuss.

FASCS is intended to be used for developing SC program families with the following characteristics: *i*) The programs are developed by end users; *ii*) The programs solve problems modeled by continuous mathematical equations; *iii*) The solutions to the equations are usually approximated and cannot be obtained without the help of a computer; *iv*) When it exists, there is a unique true solution. Although the focus is on families with the above characteristics, with some changes, FASCS can be used to develop programs that do

not have all of the above characteristics. For instance, a mesh generation program family, which does not use continuous mathematical equations to model the problem and does not have a unique true solution, can also be developed as a program family by slightly changing the Commonality Analysis stage of the methodology (Yu, 2007). Similarly, for a commercial SC program, as opposed to an end user developed program, FASCS can be extended by adding an economic analysis, which is absent for FASCS, into the methodology. More details on further exploration of the scope of FASCS are listed in the future works section of this thesis (Chapter 8).

FFEMP, as introduced in Section 2.4, is used to illustrate how FASCS can be used to develop an SC program family and how FASCS can improve the quality of SC software.

FASCS includes processes, methods and techniques for developing SC program families. It uses generic and generative approaches and provides a problem solving environment for developing SC program families. An overview of FASCS is illustrated in Figure 3.1. Similar to the general program family approach that was described in Section 1.2 (Figure 1.1), the process of FASCS has two phases: Domain Engineering and Application Engineering. Each phase has four stages. Each stage, except the Application Testing stage, provide artifacts to some other stages and each stage, except the Domain Requirement Engineering stage, provides some feedback to other stages. The major difference is that FASCS does not have a product management stage, which deals with the economics and the scope of the program family. The reason is that FASCS is intended for end user developed software and the economics of the

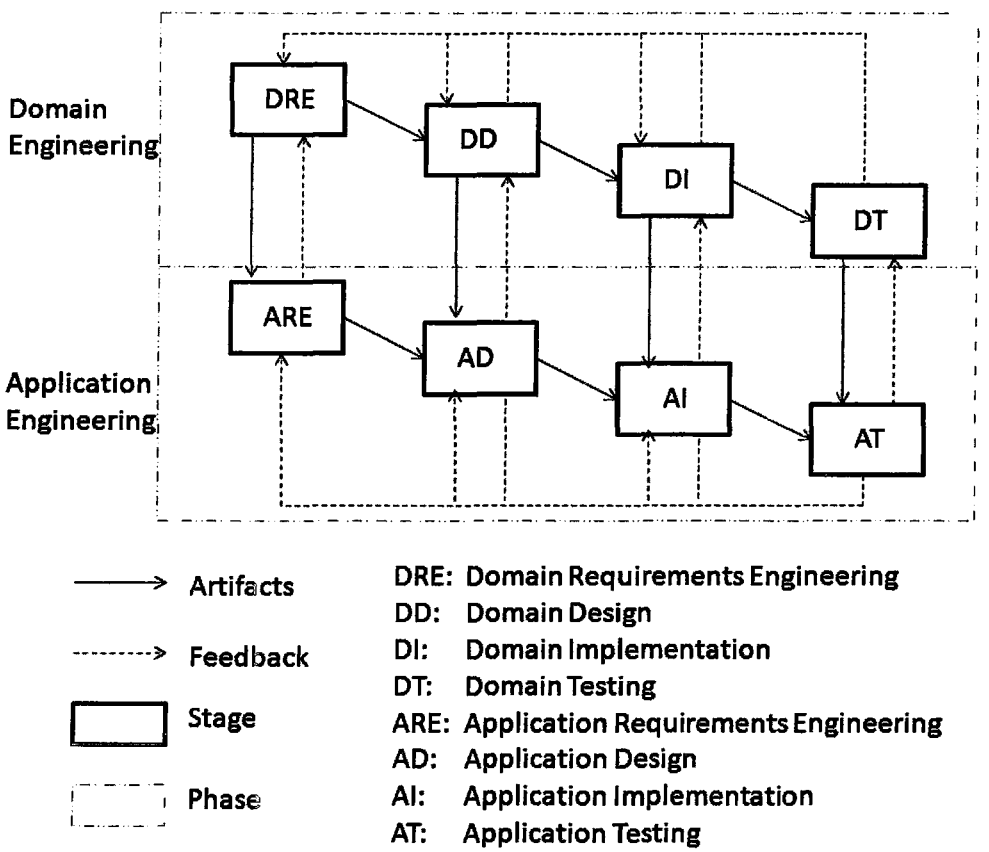

Figure 3.1: An Overview of FASCS

project are often not emphasized. The scope is identified during the domain requirements engineering stage. If it is necessary, the economic analysis can be added either as a separate stage in the domain engineering phase, or as a part of the domain requirements engineering stage. The method for analyzing the economics of the program family depends on the characteristics of the application domain and will not be further explored in the current work.

The FASCS process is iterative. The development of each stage is iterated according to the feedback from later stages. This feedback is denoted by dashed arrows in Figure 3.1. The sequence of the stages in the picture, which is indicated by solid arrows, gives the artifacts needed for the development of each stage. The sequence can also provide a rational order that users of the document will follow when reading the documentation. The process of FASCS is discussed in the rest of this chapter.

3.1 Domain Engineering

Artifacts that are common to all family members are developed in this phase. There are four stages: Domain Requirements Engineering, Domain Design, Domain Implementation and Domain Testing, as shown in Figure 3.1.

3.1.1 Domain Requirements Engineering

Domain requirements engineering, which is also called commonality analysis, provides all common and variable requirements. A new methodology, called Goal-Oriented Commonality Analysis (GOCA), is proposed for the common-

ality analysis stage in FASCS. GOCA will be described in Chapter 4. The ultimate artifacts of this phase is documentation of common and variable requirements. However, goals, theoretical models and computational models are also documented for potential reuse in the future. The definitions of the goals and models mentioned above will be given in Chapter 4. The documents for GOCA, including the Common and Variable Requirement Specification (CVRS), Theoretical Model Specification (TMS) and Computational Model Specification (CMS), are specified in Chapter 7.

3.1.2 Domain Design

The domain design in FASCS provides a reference architecture for the program family and the design of a Family Member Development Environment (FMDE).

3.1.2.1 Reference Architechture

In FASCS, the reference architecture is obtained by decomposing the program family into modules using the information hiding principle (Parnas, 1972). The modular design can improve understandability, which in turn can improve reusability and maintainability. For a program family, there are two types of potential change. One type, which is called an anticipated change, does not occur in the present, but it may happen in the future. This type of change is associated with the evolution of the program family. Another type is the variable requirements that differentiate family members. In FASCS, the module decomposition should minimize the number of related modules to be

modified for both types of change.

The module decomposition is the same for all members of the program family. It is documented in a Reference Module Guide (RMG). The module hierarchy for FFEMP is shown in Table 3.1. Detailed information on the module decomposition of FFEMP can be found in the RMG for FFEMP (Yu, 2010d).

Level 1	**Level 2**	**Level 3**	**Level 4**
Hardware-Hiding Module	File Module		
	Device Interface Module	Keyboard Input Module	
		Screen Display Module	
Behavior-Hiding Module	Input Module		
	Output Module		
	Control Module		
Software Decision Module	Data Module	Constant Module	
		Numerical Data Module	Vector Module
			Matrix Module
			Sparse Matrix Module
	Local Module	General Point Module	Node Module
			Integration Point Module
		Element Module	
		Boundary Element Module	
	Global Module	Mesh Module	
		FEM	
		Linear Solver Module	

Table 3.1: Module Hierarchy for FFEMP

The reference interfaces of the modules should also be included in the reference architecture. The major part of the interface of a module is the interfaces, including the syntax and semantics, of its access routines. The ref-

erence interfaces are documented in a Reference Module Interface Specification (RMIS). The RMIS for FFEMP can be found in Yu (2010e). Both documents (RMG and RMIS) for the reference architecture of a program family will be further discussed in Chapter 7.

The interfaces of the modules without the influence of variabilities are the same for all family members. On the other hand, the interfaces of the modules with the influence of variabilities may or may not be the same for some family members. For example, an access routine in the Element Module (M_Elm) for FFEMP, calStress, which calculates the stress, may not exist for some family members, since whether or not stress is calculated is a variability of FFEMP. Moreover, the interfaces with the same syntax may have different semantics. For instance, the access routine calConst in the above M_Elm module, which calculate the constitutive matrix for the element, has the same syntax for all family members. However, it has different semantics depending on the stress state and strain state, which are two variabilities of FFEMP. These two variabilities and the access routine calConst will be discussed later in this chapter (Section 3.2.2) and in Chapter 7. Details on the variabilities that impact the interfaces of modules for FFEMP can be found in Yu (2010e).

In FASCS, modules should be designed to maximize the number of modules that have the same interface for all family members to improve reusability. In addition, modules that have different interfaces for some family members should provide information for developing the interfaces for a specific family member. For the above example access routine calConst, the MIS for FFEMP (Yu, 2010e) not only gives the general property that the constitutive matrix

$\mathbf{D}$ possesses, which is $\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\epsilon}$ (Equation 2.1b), as mentioned in Section 2.4, it also provide the formula of $\mathbf{D}$ for a general 3D domain and instructions for simplifying the general formula according to the values of the variabilities stress state and strain state. More details on the formulas and instructions can be found in Section 3.2.2 and Section 7.2.2.

3.1.2.2 Family Member Development Environment

A Family Member Development Environment (FMDE) provides a facility for application engineers to quickly develop a specific member of a program family. It contains:

- a Domain Model (DM) that defines a language to describe a specific family member,
- a Family Member Generation Process (FMGP) that gives instructions on how to generate a specific family member using the environment,
- a Variable Code Generator (VCG) that generates variable code,
- a Test Case Generator (TCG) that generates the variable part of some benchmark test cases, and
- a Family Member Assembler (FMA) that assembles code and test cases for a specific family member.

The application engineers follow the instructions given in FMGP. They write a program in the language defined in the DM to specify the member of a program family they desire and they use FMA. The FMA calls the VCG and TCG to

generate appropriate variable code and the variable part of the benchmark test cases and then the FMA assembles the variable code with common code and the variable part with the common part of the test cases. The common code and the common part of the test cases are developed in the domain engineering stage. The use of an FMDE addresses the modification challenge introduced in Section 2.2.1 since developing a new member of a program family by using FMDE is easier than developing a new single program or modifying an existing single program. It can also improve reusability and reliability, as will be discussed in Chapter 5.

The idea of using an environment to help application engineers generate family members is presented by Weiss and Lai (1999). However, unlike FASCS, there is only one environment in Weiss and Lai (1999). Dividing the environment into sub-environments can make the environment easier to use. Another difference between the environment in FASCS and that in Weiss and Lai (1999) is the Test Case Generator (TCG), which is not included in Weiss and Lai (1999). Including TCG can improve the reusability and reliability, as Chapter 5 will discuss. Moreover, Weiss and Lai (1999) did not give a specific name to their environment. More details on FMDE are given in Chapter 5.

3.1.3 Domain Implementation

During the domain implementation stage, code that is common for all family members is developed. The VCG, which was designed in the domain design stage, is implemented. The part of the FMA that relates to code is implemented. More details on Domain Implementation can be found in Section 6.1.

FASCS suggests using tools, such as version control, to help with the implementation. For example, FFEMP used subversion to maintain current and historical versions of code. The subversion was also used for developing the documents for FFEMP.

3.1.4 Domain Testing

The TCG that was designed during the domain design stage is implemented during the domain testing stage. The remaining part of the FMA, which relates to the test cases, is implemented at this stage.

The test cases in FASCS include:

- benchmark test cases for testing common code in variable routines,
- benchmark test cases for testing nonfunctional requirements for a specific family member,
- test cases for testing common routines, and
- test cases for testing variable routines.

Except for test cases for testing common routines, each set of the above test cases includes common and variable parts. The test cases for testing common routines and common parts of other test cases are developed in the domain testing stage using a traditional approach. The variable part for benchmark test cases for testing common code in variable routines are automatically generated using TCG and tests for common code in variable routines are performed.

Tests of the routines are also performed. More details on testing can be found in Section 6.2.

When possible, unit testing tools, such as CppUnit (CppUnit, Retrieved 2010) for testing C++ code and JUnit (Beck, Retrieved 2010) for testing Java code, should be used for helping developers perform the testing. Since FFEMP is developed using C++, CppUnit is used for its testing.

3.1.5 Traceability Matrix

Traceability matrices should be developed during the domain engineering phase. These traceability matrices include the artifacts for all stages in the domain engineering stage. The details are given in Chapter 7, where the documentation of FASCS is presented.

3.2 Application Engineering

Since artifacts developed in the domain engineering stage can be reused, developing a member of a program family in the application engineering stage is simpler than developing a single program. The application engineering phase includes four stages: Application Requirements Engineering, Application Design, Application Implementation and Application Testing, as shown in Figure 3.1.

3.2.1 Application Requirements Engineering

During the application requirements engineering stage, the value for each variable requirement is determined. The requirements for a specific family member consist of the common requirements that are developed in the domain requirement engineering stage and the determined variable requirements.

The determined variable requirements are documented in the Specific Variable Requirement Specification (SVRS). It is a very simple document and only contains a list of determined variable requirements. SVRS, together with Common and Variable Requirement Specification (CVRS) that was developed in the domain requirement engineering stage, are all the documents needed to document the requirements for a given family member. An example of the determined variable requirements for a member of FFEM is shown in Figure 3.2.

FVR_ElmShape = TRI
FVR_NumNode = 3
FVR_IntMethod = GAUSSQ
FVR_NumIpts = 4
FVR_StressS = $\langle F, F, T, F, T, T \rangle$
FVR_StrainS = $\langle F, F, F, F, F, F \rangle$
FVR_NumBNode = 2
FVR_BIntMethod = GAUSSQ
FVR_NumBIpts = 2
FVR_Stress = F
FVR_Strain = F

Figure 3.2: Determined Variabilities for a Member of FFEMP

The family member shown in Figure 3.2 can solve for plane stress problems, which is determined by the stress state (FVR_StressS = $\langle F, F, T, F, T, T \rangle$)

and the strain state (FVR_StrainS = $\langle F, F, F, F, F, F \rangle$). The domain is decomposed into triangular elements (FVR_ElmShape = TRI). Each element has 3 nodes (FVR_NumNode = 3) and each boundary has 2 nodes (FVR_NumBNode = 2). Both the stiffness matrix and the consistent load vector are calculated using Gauss Quadrature (FVR_IntMethod = GAUSSQ and FVR_BIntMethod = GAUSSQ). The number of integration points is 4 for calculating the stiffness matrix (FVR_NumIpts = 4) and the number of integration points is 2 for calculating the consistent load vector (FVR_NumBIpts = 2). This particular family member does not calculate and output the stress and strain value (FVR_Stress = F and FVR_Strain = F).

The above variable requirements for FFEMP are easy to understand except for FVR_StressS and FVR_StrainS. However, to understand the meaning and format of these two variable requirements, one requires to understand the definitions of stress and strain, which are two concepts in mechanics. These definitions are complicated and giving the definitions in this stage would disrupt the flow of the presentation. Hence, more detailed information on FVR_StressS and FVR_StrainS is not given in this section. It can be found in Chapter 4 and 7 and in Yu (2010e).

3.2.2 Application Design

Since the architectures for all members of a program family are the same, the Reference Module Guide (RMG), which was developed in the domain design stage, can be reused to specify a specific family member. The Reference Module Interface Specification (RMIS), which was also developed in the do-

main design stage, can also be reused, since it documents the common part of the interfaces. Interfaces that are different for some family members are developed in the application design stage. The variable interfaces are documented in the Specific Module Interface Specification (SMIS). Because the RMIS provides information on how to develop the specific module interface, as discussed in Section 3.1.2, the development of variable module interfaces is relatively straightforward. Documentation for the architecture of a specific family members contains the RMG, RMIS and SMIS.

As an example, one can take the development of the semantics for the access routine calConst in the M_Elm module of FFEMP, which was mentioned in Section 3.1.2. The generic expression for the linear elastic constitutive matrix for a general 3D problem $\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\epsilon}$ is used in the Reference Module Interface Specification (RMIS). The detailed expression for $\mathbf{D}$ is shown in Equation 3.1 with E as the elastic modulus and ν as the Poisson's ratio.

$$\mathbf{D} = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} * \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & 1 & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \tag{3.1}$$

In the SMIS for the family member in Figure 3.2, the semantics of calConst is refined, according to the information specified in the RMIS, to a 3×3 matrix as shown in Equation 3.2.

$$\mathbf{D} = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \tag{3.2}$$

Details on how Equation 3.1 is simplified to Equation 3.2 can be found in Chapter 7 and Yu (2010e).

3.2.3 Application Implementation

The deliverable code for a family member is an assemblage of common and variable code. The common code was developed in the domain implementation stage and the variable code is developed in the application implementation stage. The common and variable code are assembled in the application implementation stage.

Since the Family Member Development Environment (FMDE) was developed in the domain engineering phase, the variable code can be generated using the Variable Code Generator (VCG) and the common and variable code can be assembled using the Family Member Assembler (FMA). The guidance of using VCG and FMA can be found in the Family Member Generation Process (FMGP). As mentioned, details of FMDE are presented in Chapter 5 and the details of the implementation of a program family are discussed in Section 6.1.

3.2.4 Application Testing

Variable parts of test cases for testing the specific family member are developed and the tests for the specific family members are performed during the application testing stage. The common parts of these test cases were developed in the domain testing stage, as Section 3.1.4 mentioned. The variable parts include the variable part of benchmark test cases for testing nonfunctional requirements and variable part of test cases for testing variable routines. The variable part of benchmark test cases for testing nonfunctional requirements is automatically generated using the Test Case Generator that was developed in the domain engineering stage. The variable part of test cases for testing variable routines may or may not be generated automatically depending on how complicated the test cases are. Complete test cases are assembled using the Family Member Assembler that was developed in the domain engineering stage. More details of the test of a program family are discussed in Section 6.2.

Chapter 4

Goal Oriented Commonality Analysis

Goal Oriented Commonality Analysis (GOCA) is a process and methodology to elicit, analyze and document common and variable requirements for SC program families. Members of these program families can solve problems that are modeled as continuous mathematical equations. The motivation of proposing GOCA is to improve SC software quality factors, such as reusability and reliability, and to address the technique selection challenge introduced in Section 2.2.1.

An overview of GOCA is illustrated in Figure 4.1. In this figure, a rectangle represents artifacts including Functional Goals (FG), Nonfunctional Goals (NG), the Theoretical Model (TM), the Computational Model (CM), Common Requirements (CR) and Variable Requirements (VR). The explosion shape represents the Terminology Definitions (TD). TD includes Theoretical

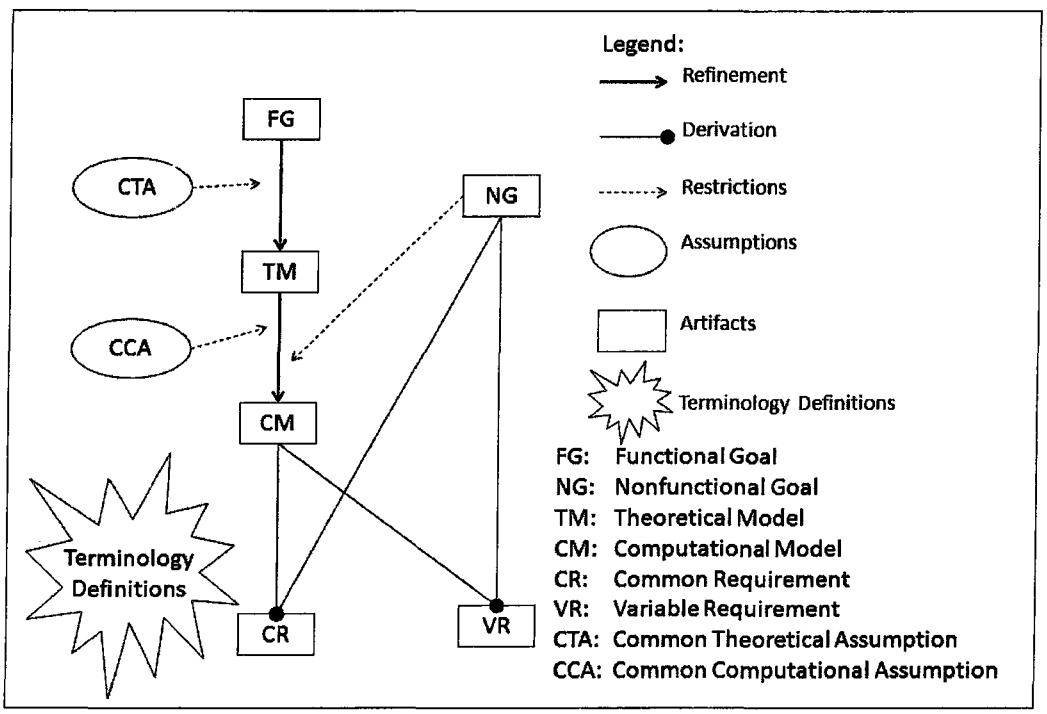

Figure 4.1: Goal Oriented Commonality Analysis

Terminology Definitions (TTD) and Computational Terminology Definitions (CTD). An oval represents assumptions, such as Common Theoretical Assumptions (CTA) and Common Computational Assumptions (CCA). There may also be Variable Assumptions (VTA) to refine the theoretical and computational terminology. All of the above GOCA terminology will be defined when the process is explained in detail later in this chapter.

Solid arrows represent refinement relations. For example, theoretical models refine functional goals. A solid line with a dot at one end represents a derivation relation. For example, some common requirements are derived from computational models. The derivation associates with requirements, while the refinement does not. The dashed arrows represent applying constraints to a refinement. For example, both common computational assumptions and nonfunctional goals are applied when theoretical models are refined to computational models.

GOCA is based on the fact that most scientific problems can be modeled as theoretical models, which are composed of mathematical equations. These equations are often continuous. If the problems are complicated, solving these equations becomes difficult without the help of a computer. However, a computer usually cannot directly solve continuous equations due to its discrete nature. Other sets of equations that approximate theoretical models, which can be solved discretely by a computer, are required and these sets of equations form the computational models.

The purpose of the commonality analysis is to obtain common and variable requirements, which are the basis for building a program family. The de-

tails of the process are specified in the remainder of this chapter. The program family FFEMP, which was introduced in Section 2.4, is used as an example to illustrate how GOCA can be used to develop artifacts in the commonality analysis stage.

The documentation of the commonality analysis using GOCA is provided in three documents, namely Common and Variable Requirement Specification (CVRS), Theoretical Model Specification (TMS) and Computational Model Specification (CMS). The purpose of the separation is to improve reusability. Since two of the documents, TMS and CMS, are abstract, they are highly, sometimes completely, reusable.

The details on documentation for the commonality analysis using GOCA are presented in a separate chapter (Chapter 7). GOCA and the documentation of GOCA are presented in separate chapters because the abstraction that facilitates separating TMS and CMS leads to the interleaving of some artifacts according to the sequence of the presentation in this chapter. For example, the Theoretical Terminology Definitions (TTD) should be documented together with the Theoretical Model (TM) since the definitions are mainly used for this model. These definitions should be abstract in TMS to make the specification reusable. However, some of these definitions are used in the computational model and requirements. The abstract definitions should be refined for the specific computational model and requirements. The refinement should be documented in the corresponding CMS and CVRS. If these complicated relations and how to document the different level of the definitions were introduced together in this chapter, the flow of the presentation would

be interrupted.

Another reason for separating the documents is to emphasize the importance of the documentation for FASCS. Moreover, introducing all documents for the domain engineering phase together give the user an idea of what documents are included in the domain engineering phase.

The process of GOCA starts from identifying goals (Section 4.1). There are two types of goals: functional goals and nonfunctional goals. Functional goals are refined to a theoretical model (Section 4.3) and the theoretical model is refined to a computational model (Section 4.4). Common and variable requirements (Section 4.5) are derived from the computational model, as well as the nonfunctional goals. During the refinements, assumptions are applied. Since they are strongly related to the theoretical model and computational model, these assumptions, which include theoretical assumptions and computational assumptions, are defined when corresponding models are discussed. Terminology Definitions (Section 4.2) are given explicitly to avoid ambiguity. A summary of the common and variable requirements for the example program family, FFEMP, and a discussion on the scope of GOCA are given at the end of this chapter (Section 4.6).

4.1 Goals

A goal is the starting point of the GOCA process. A goal represents a real world problem to be solved by the program family. It is defined as follows:

Goal A goal is an abstract objective that the system under consideration

should achieve.

Goals have been used in requirement engineering for a long time. A goal is defined in van Lamsweerde (2001) as "*capturing, at different levels of abstraction, the various objectives the system under consideration should achieve.*" In van Lamsweerde (2001), high level goals are refined to subgoals to help with eliciting requirements. However, SC software often has a well defined target or objective and the process described by van Lamsweerde (2001) would complicate most SC software development. Experience shows that goals usually do not have to be refined in SC software (Smith and Lai, 2005; Smith, 2006). In GOCA, a goal is the highest level of abstraction and is not refined to subgoals.

For some SC problems, goals, which are the most abstract objectives, may not be as abstract as those for other non SC programs. These objectives may be classified as pre-requirements for these non SC programs. However, these objectives are classified as goals in this research because the author would like to reserve the term *Requirement* to be as formal as possible. As it will discuss later, mathematical equations are used in the documents for SC programs. These equations are easily misinterpreted. The formalization can avoid any unnecessary ambiguity, which can improve reliability.

A goal can be functional or non-functional. The definitions of functional goals and nonfunctional goals are given below:

Functional Goal (FG): Functional goals specify services the program family provides.

Nonfunctional Goal (NG): Non-functional goals specify the quality of the services the program family provides.

An example functional goal and an example nonfunctional goal for FFEMP are given below.

FG1 (FG_Displacement): Given the computational domain, the material properties and the boundary conditions, FFEMP can solve for the displacement of any point in the domain.

NG1 (NG_Accuracy): The members of FFEMP should have a certain level of accuracy.

As the examples illustrate, goals are expressed in natural languages for improving understandability. However, natural language is ambiguous. The ambiguity for goals is allowed because goals will ultimately be refined to common and variable requirements. As long as the requirements are unambiguous, quality factors for a program family, such as accuracy and testability, will not be decreased. Theoretically, all quality factors of software should be included in the the list of nonfunctional goals. However, emphasis should be given to the quality factors that are particularly important to the family being developed.

4.2 Terminology Definitions

As mentioned in Section 4.1, goals may be ambiguous, but the ultimate requirements must be unambiguous. Hence, unambiguous terminology is needed to describe the models and the requirements. However, some of the terminology does not have standard definitions and notations. Different people may use

different names and different notations. For example, stress is represented by σ in Bauld (1986), but it is represented by χ in Southwell (1941). Furthermore, the symbol σ has many other meanings besides stress, such as the standard deviation. Explicitly defined terminology and notations can avoid ambiguity, which implies that the accuracy of the software, the usability of the document and the understandability of the whole system can be improved.

There are two types of terminology, namely theoretical terminology and computational terminology. Theoretical terminology is used to express the theory underneath the model that describe the problem to be solved. Computational terminology, such as the *stiffness matrix* in FFEMP, is used to specify computational models and related issues. However, some theoretical terminology can be refined to computational terminology. For example, *stress* and *strain* are general theoretical concept in FFEMP, but they are refined based on the decisions related to computational issues. The theoretical terminology *stress* is defined as stress anywhere in the computational domain. However, the computational terminology *stress* is defined as stress for the nodes. The definitions of both theoretical terminology and computational terminology are necessary for unambiguous requirements.

Terminology definitions may be refined by introducing assumptions. However, the essential part of the definition, including the units and the symbol to denote the definition, remains the same; that is, part of the definition is common to all family members. For example, *stress* is defined as the force per unit area associated with different directions at a point within a body. A detailed definition can be found in the Theoretical Model Specification for

FFEMP (Yu, 2010b). The symbol to represent stress is $\boldsymbol{\sigma}$ and the units are $L^{-1}Mt^{-1}$, where units are given in terms of the mass (M), length (L) and time (t). The stress can be represented by a tensor as in Equation 4.1:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix} \tag{4.1}$$

However, this representation for stress is not well suited for efficient computation. After introducing common theoretical assumptions for FFEMP, such as using a rectangular Cartesian coordinate system and no distributed moments or couples stresses, some components in the tensor can be eliminated. Instead of a tensor, a vector is used to represent the stress, because it removes redundant information and a vector is a commonly accepted representation for stress in the computational mechanics community. The use of a commonly accepted representation can improve the understandability, and hence improve reusability. The refined representation is shown in Equation 4.2:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{bmatrix} \tag{4.2}$$

Equation 4.2 can be refined to Equation 4.3 when the assumption of

two dimensional domain is applied.

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \tau_{xy} \end{bmatrix} \tag{4.3}$$

Equation 4.2 is a general 3D format of stress and can be reused in many solid mechanics problems. On the other hand, Equation 4.3 is less general and can only represent stress in a 2D domain.

4.3 Theoretical Model

A theoretical model represents a real world problem (goals). It is defined as:

Theoretical Model (TM): A theoretical model, with respect to the goals, is a set of mathematical equations that refines the goals.

The theoretical model for FFEMP was shown in Equation 2.1 when FFEMP was introduced in Section 2.4.

The theoretical model can refine the goals because of theoretical assumptions that are applied so that the underlying theories hold. Below is the definition of theoretical assumptions.

Theoretical Assumption (TA) Theoretical assumptions, with respect to goals and a theoretical model, are a set of hypotheses such that when these hypotheses hold, the theoretical model can correctly refine the goals.

Theoretical model for many SC problems have been developed for years and basic theories underlying the models rarely change. Hence, GOCA defines common and variable assumptions to improve reusability. The definition of common theoretical assumption is given below.

Common Theoretical Assumption (CTA) Common theoretical assumptions with respect to goals and a theoretical model are a minimal set of theoretical assumptions, such that when these common theoretical assumptions hold, the theoretical model captures the goals.

Since the set is minimal, this definition eliminates any assumptions that are unnecessary for a theoretical model to represent goals. In fact, the theoretical model, itself, is general and can be reused. For instance, the theoretical model for FFEMP (Equation 2.1) can be used for solving elasticity problems in solid mechanics in any dimension. The scope of the theoretical model is restricted by the definitions of the terminology that are used by the model. As discussed in Section 4.2, the definition for stress is Equation 4.2 for a general 3D domain. However, it can be refined to Equation 4.3 when the assumption of a 2D domain holds. The scope of the theoretical model (Equation 2.1) is larger when the stress is defined by Equation 4.2 than when the stress is defined by Equation 4.3.

The common theoretical assumptions can be reused as well. For example, one of the common theoretical assumptions for FFEMP is that dynamic effects are excluded. This assumption holds for any static problem. In addition, the theoretical terminology definitions with respect to common theoretical assumptions are general, as the example of the definition for stress shows.

Hence, these definitions can be reused.

Some program families may not have the same scope as the largest scope for the theoretical model. Additional assumptions need to be introduced to restrict the scope. These assumptions are variable assumptions and are defined as follows:

Variable Assumption (VA) Variable assumptions with respect to goals and a theoretical model are a set of variable properties that the problems to be solved possess. Each property in the set has certain possible values. Once the possible values are determined or partially determined, the scope of the theoretical model is determined.

For example, VTA_SelfWeight is one of the variable assumptions for FFEMP. There are two possible values, T (true) and F (false). If VTA_SelfWeight $= T$, then the self weight is included in the calculation; otherwise, if VTA_SelfWeight $= F$, then the self weight is not included in the calculation.

Variable assumptions do not directly impact the theoretical model. However, they may impact the refinements of some terminology definitions. For example, if the dimension of the computational domain were an assumption for FFEMP, it would be a variable assumption because it does not directly impact the computational model for FFEMP (Equation 2.1). However, it would impact some terminology definitions. If the assumption is that the computational domain is 2D, the definition of the theoretical terminology stress would be refined from Equation 4.2 to Equation 4.3.

Although the representation of the theoretical model, and the representation of the computational model that will be discussed in Section 4.4, are

the same, the scopes of the models are changed by introducing variable assumptions because the definitions of some terminology used in the models are changed. The representations of the theoretical model and the computational model depend on common assumptions.

A variable assumption may or may not be determined. It also can be completely or partially determined. If it is not determined completely, it is still a variability that can either be set later, so that it becomes a common requirement of the program family, or it can remain a variability and becomes a variable requirement of the program family. For example, the variable assumption VTA_SelfWeight $= F$ for FFEMP is completely determined. On the other hand, VTA_StressS and VTA_StrainS are not determined and become the variabilities FVR_StressS and FVR_StressS. VTA_StressS and VTA_StrainS represent stress state and strain state, respectively. The possible values for the two variable assumptions are a sequence of 6 booleans. If an entry in VTA_StressS is equal to T, this means that the corresponding entry in the stress vector (Equation 4.2) is equal to zero. The relation between VTA_StrainS and the strain vector is similar.

The sources of variabilities can be traced by explicitly specifying the variable assumptions. The reusability of the whole system is improved because the artifacts can be reused when the values of variable assumptions are changed. For instance, if a program family for 1D computational domain is required, then all of the stress components except one are equal to false and all of the strain components are equal to false. Therefore, one can change VTA_StressS and VTA_StrainS to be VTA_StressS $= \langle F, T, T, T, T, T \rangle$ and

VTA_StrainS = $\langle F, F, F, F, F, F \rangle$. The rest of the artifacts, including the theoretical model, terminology definitions and assumptions, remain the same and can be reused.

Variable assumptions only relate to theoretical issues. Any assumption relating to computational issues should not be decided in this stage, to keep the theoretical model and the computational model abstract. The abstraction can improve the reusability of the document. These computation related assumptions should be determined later as a common or variable requirement.

4.4 Computational Model

A computational model refines a theoretical model. This refinement is restricted by both common computational assumptions and nonfunctional goals.

4.4.1 Definitions

The definition of computational model is given below.

Computational Model (CM): A computational model, with respect to a theoretical model, is a set of mathematical equations that approximate the theoretical model in a form that can be solved by a computer.

The relative error between the solutions of the computational model and the theoretical model should be within a certain range. The specific range is determined by the nonfunctional goals related to accuracy and precision. An example computational model, which approximates the computational model for FFEMP (Equation 2.1), is expressed as Equation 2.2 in Section 2.4.

Assumptions are required for the refinement from a theoretical model to a computational model. These assumptions are called computational assumptions and are defined below.

Computational Assumption (CASS) Computational assumptions, with respect to a theoretical model and a computational model, are a set of hypotheses, such that when these hypotheses hold, the theoretical model can be refined to the computational model.

Similar to common theoretical assumptions, common computational assumptions are defined as:

Common Computational Assumption (CCA) Common computational assumptions, with respect to a theoretical model and a computational model, are a minimal set of hypotheses, such that when these hypotheses hold, the theoretical model can be refined to the computational model.

As for a theoretical model, a computational model is abstract and can be reused due to the use of common computational assumptions. For example, the computational model for FFEMP (Equation 2.2) can be reused by any FEM program. However, some computational terminology needs to be redefined to change the scope of the computational model.

Although the reuse of common computational assumptions does not occur as frequently as the reuse of a computational model, the common computational assumption still can be reused by another program. For example, the common computational assumptions for FFEMP can be reused by programs that solve elasticity problems in solid mechanics using FEM.

The primary common computational assumption for a theoretical model is the technique for solving the theoretical model, which is represented by a set of equations. There may be more than one technique available and the selection of the technique may be affected by quality factors. As mentioned in Section 2.2.1, the systematic selection of the appropriate technique is the technique selection challenge for SC software. GOCA addresses this challenge and adopts a decision making technique for selecting the appropriate technique. This technique is based on the ranking of nonfunctional goals for the program family and will be discussed next.

4.4.2 Ranking Nonfunctional Goals

GOCA uses the Analysis Hierarchy Process (Saaty, 1980, 2008) to rank nonfunctional goals to choose the most appropriate technique to solve the theoretical model. This technique can also be used for other decisions related to nonfunctional goals or nonfunctional requirements, such as the selection of algorithms for solving system of equations, based on nonfunctional requirements, when the program family is implemented.

AHP provides a comprehensive and rational framework for evaluating alternative solutions. It has been used by software engineers in requirement analysis (Kott et al., 1996) and testing (McCaffrey, 2005). AHP is also used in the development of SC software for ranking the relative priority of the nonfunctional requirements (Smith, 2006).

The decision making process adopted by GOCA is presented below. Applying this process requires some knowledge of AHP. Fortunately, tools,

such as Init (Retrieved December 2010), are available. FASCS suggests using these tools to do the calculation. The advantage is that the domain engineers can use the process without dealing with detailed calculations.

The steps for selecting the appropriate technique to solve the theoretical model for a program family according to the related nonfunctional goals are given below.

1. List techniques considered by the program family to solve the theoretical model.

2. List nonfunctional goals related to the selection of the technique.

3. Construct a pairwise comparison matrix between nonfunctional goals to obtain the priorities from the matrix.

4. Construct a set of comparison matrices between techniques with respect to nonfunctional goals and obtain the priority of each technique with respect to each nonfunctional goal from the matrices.

5. Calculate the overall priorities of the techniques from the priorities of nonfunctional goals that are calculated from the Step 3 and the priorities of techniques with respect to nonfunctional goals that are calculated from Step 4.

An illustration of how the process is used to select the technique to solve the theoretical model (Equation 2.1) for FFEMP is presented below. The priorities calculated for the example use Init (Retrieved December 2010). More

details on the example can be found in the Computational Model Specification of FFEMP (Yu, 2010c).

1. The list of available techniques that are considered for FFEMP are:

   (a) Direct Method (DM)

   (b) Finite Difference Method (FDM)

   (c) Finite Element Method (FEM)

   where, DM means using the closed form solution.

2. The list of related nonfunctional goals are:

   (a) NG_Accuracy

   (b) NG_Precision

   (c) NG_Efficiency

   (d) NG_Reusability

3. The pairwise comparison matrix between nonfunctional goals is:

Quality Preference	Accuracy	Precision	Efficiency	Reusability
Accuracy	1	1	1	1/5
Precison	1	1	1	1/5
Efficiency	1	1	1	1/5
Reusability	5	5	5	1

According to Saaty (1980), this table gives the importance of one nonfunctional goal over another for FFEMP. For example, 5 in the (Reusability, Accuracy) entry (last row second column) means that Reusability is

strongly (5 times) more important compared to Accuracy for FFEMP. The numbers in this matrix, such as 5, are determined by the domain engineers.

4. The technique comparison matrix with respect to reusability is:

Reusability	DM	FDM	FEM
DM	1	1/7	1/9
FDM	7	1	1/2
FEM	9	2	1

The other three matrices are similar. These matrices give how well one technique compares to another one to solve Equation 2.1 with respect to the corresponding nonfunctional goals. For example, 9 in the (FEM, DM) entry means that FEM is extremely (9 times) more reusable comparing to DM. The numbers, such as 9, in the matrices are obtained by consulting domain experts on solving Partial Differential Equations. FEM is the most reusable because it is superior at solving problems over an irregular computational domain. DM is the least reusable since only a few problems have closed form solutions. The numbers are heuristic and subjective. The major contribution is that this example shows the possibility of quantifying nonfunctional goals.

5. The calculated overall priorities of the techniques are:

DM	0.2347
FDM	0.3042
FEM	0.4611

From the above result, FEM has the highest priority. Hence, FEM is select to solve Equation 2.1. However, if the importance of nonfunctional goals changes, the most appropriate technique for solving Equation 2.1 may change. For example, if the pairwise comparison matrix in Step 3 emphasizes accuracy to become:

Quality Preference	Accuracy	Precision	Efficiency	Reusability
Accuracy	1	9	9	9
Precison	1/9	1	1	1
Efficiency	1/9	1	1	1
Reusability	1/9	5	5	1

then the overall priorities change to

DM	0.6934
FDM	0.1637
FEM	0.1428

The DM become the most appropriate technique. The other part of the process would stay the same.

4.5 Common and Variable Requirements

Common and variable requirements for the program family are derived from computational models and nonfunctional goals. Both common and variable

requirements can be functional or nonfunctional.

The source of nonfunctional requirements, including Nonfunctional Common Requirements (NCRs) and Nonfunctional Variable Requirements (NVRs), is the nonfunctional goals. The major source of functional requirements, including Functional Common Requirements (FCRs) and Functional Variable Requirements (FVRs), is the computational model. However, some nonfunctional goals may also need to be fulfilled by functional requirements. For example, the nonfunctional goal NG_Reusability is a nonfunctional goal stating that all member of FFEMP should have a certain level of reusability. This nonfunctional goal is fulfilled by using FEM to solve the theoretical model represented by Equation 2.1, which is a functional requirement for FFEMP. Any terminology in the functional requirements, including FCRs and FVRs, should be defined in the corresponding terminology definition sections.

4.5.1 Common Requirements

Functional common requirements specify what all members of the program family should or should not do. They are similar to functional requirements for a single program. An example functional common requirement is shown in Table 4.1. This requirement specifies that any member of FFEMP can solve the displacement using Equation 4.4. This equation has been shown in Section 2.4. It is redisplayed in this example for ease of reference. More details on each field of this table and other tables in this section (Section 4.5) are given in Section 7.1.1 when the documentation of common and variable requirements is discussed.

Number	**FCR1**
Label	**FCR_Displce**
Related Items	TTD_Elasticity, TTD_Young, CTD_Mesh, CTD_Displace, CTD_Stiff, CTD_TractionB, CTD_DisplaceB, CTD_Load, CM
Description	FFEMP can solve for displacement on each node of the meshed computational domain, using the Finite Element Method, given the material properties (E, ν), the mesh **M**, including boundary conditions. The formula is as in Equation 4.4. $$\mathbf{F} = \mathbf{Ka} \quad (4.4)$$
History	Created – Oct., 2009

Table 4.1: An Example Functional Common Requirement for FFEMP

Nonfunctional goals require all members of the program family to meet the minimum requirements for the corresponding quality factors. These minimum quality requirements form nonfunctional common requirements. An example nonfunctional common requirement is shown in Table 4.2.

Number	**NCR3**
Label	**NCR_Reliability**
Related Items	NG_Reliability
Description	All members of FFEMP should at least have the same level of reliability as OOFEM (Patzák and Bittnar, 2001b).
History	Created – July 2010

Table 4.2: An Example Nonfunctional Common Requirement for FFEMP

Nonfunctional requirements are difficult to specify because they are difficult to quantify. Instead of arbitrarily choosing a value, GOCA suggests comparing nonfunctional common requirements with some existing programs that solve the same problem. For example, reliability for FFEMP can be specified as having all members of FFEMP be at the same level of reliability as that of OOFEM (Patzák and Bittnar, 2001b), as shown in Table 4.2. This requirement is still not validatable. However, it is enough at the Domain Requirement Engineering stage, since validatable benchmark test cases will be given for testing these nonfunctional requirements in the Application Engineering phase. The idea for testing nonfunctional requirements by comparing benchmark test cases with existing programs was successfully used in Yu (2007) and Smith and Yu (2009) for developing a Parallel Mesh Generation Toolbox (PMGT).

4.5.2 Variable Requirements

Functional Variable Requirements

Functional variable requirements specify what some members of the program family should or should not do. A functional variable requirement is specified as a variability and its possible values, called variants. Once the values of the variants for a functional variable requirement are determined for a family member, it becomes a functional requirement for the member. An example functional variable requirement is shown in Table 4.3.

Number	**FVR4**
Label	**FVR_NumIpts**
Type	$\mathbb{N}$
Related Items	CM, FVR_ElmShape, FVR_IntMethod
Description	The number of integration points if GAUSSQ is selected for FVR_IntMethod
Dependency	Optional
Constraints	GC_ni, RC_mi
Variants	[3 .. MIP], where MIP is a constant representing the maximum number of integration points
History	Created – October, 2009

Table 4.3: An Example Functional Variable Requirement for FFEMP

Inspired by Kang et al. (1990) and Pohl et al. (2005), GOCA classifies a functional variable requirement into mandatory or optional. A mandatory functional variable requirement turns into a requirement for all family members. However, an optional functional variable requirement can only turn into a requirement for some family member when some conditions hold. For example, the functional variable requirement FVR_NumIpts (the number of integration points) shown in Table 4.3 is optional for FFEMP. It becomes a

requirement for a member of FFEMP if and only if the functional variable requirement FVR_IntMethod (integration method) is "GAUSSQ."

Classifying a functional variable requirement into mandatory or optional can improve the reliability. For example, if a specific member for FFEMP uses the direct method to calculate the integrations in the calculation of the stiffness matrices (FVR_IntMethod = DIR), then it may confuse the application engineers who design or implement a specific family member of FFEMP to have the variability of integration points (FVR_NumIpts). Having both functional variable requirements can also increase the possibility for application engineers to make mistakes. Having FVR_IntPts may mislead them to think that the integration method is GAUSSQ because the variability FVR_NumIpts is only meaningful when FVR_IntMethod = GAUSSQ.

There are constraints between functional variable requirements. Constraints in Kang et al. (1990) and Pohl et al. (2005) are more complicated than what is needed by an SC program. Hence, GOCA defines only two types of external constraints between functional variable requirements. One type of constraint is called a Requiring Constraint. It determines if an optional requirement is required for a particular program family. Hence, at least one of the functional variable requirements in the constraint is optional. One of the constraints for FVR_NumIpts shown in Table 4.3, RC_mi, is:

$$\text{FVR\_NumIpts} \in V \iff \text{FVR\_IntMethod} = \text{GaussQ} \tag{4.5}$$

where V represents a set of variable requirements for the family member.

Another type of constraint is called a General Constraint. It is a constraint between functional variable requirements, which can be mandatory or optional. For example, in FFEMP, a constraint between the functional variable requirement FVR_ElmShape and FVR_NumNode, which is the constraint GC_ni in Table 4.3, is:

$$\begin{aligned}(\text{FVR\_ElmShape} &= \mathsf{TRI} \wedge \text{FVR\_NumNode} \geq 3) \vee \\ (\text{FVR\_ElmShape} &= \mathsf{QUAD} \wedge \text{FVR\_NumNode} \geq 4)\end{aligned}$$

meaning that the number of nodes for a triangular element is greater than or equal to 3 and the number of nodes for a quadrilateral element is greater than or equal to 4.

Nonfunctional Variable Requirements

In addition to the minimum quality requirements for all family members, some members of the program family may have higher requirements on some quality factors. These quality factors are nonfunctional variable requirements. Nonfunctional variable requirements for FFEMP are NVR_Accuracy and NVR_Efficiency, which specify different requirements on accuracy and efficiency for some family members.

In FASCS, the nonfunctional variable requirements are quantified by their priorities that are obtained by using AHP (Saaty, 1980, 2008) to rank them. For example, the variants of the nonfunctional variable requirements NVR_Accuracy and NVR_Efficiency for a member of FFEMP can be specified by pairwise comparison matrix as shown in Table 4.4. For FFEMP,

NVRs	NVR_Accuracy	NVR_Efficiency
NVR_Accuracy	1	5
NVR_Efficiency	1/5	1

Table 4.4: An Example Pairwise Comparison between Nonfunctional Variable Requirements

NVR_Accuracy is strongly more important than NVR_Efficiency. The priorities are:

NVR_Accuracy = 0.8333
NVR_Efficiency = 0.1667

The nonfunctional variable requirements specified in this stage are different than nonfunctional goals, which are specified in Section 4.4.2, although they both deal with quality aspects. Nonfunctional goals specify the differences between quality factors for program families and nonfunctional variable requirements specify the differences between quality factors for members of a program family.

These nonfunctional variable requirements are used to make decisions when a specific family member is developed. When the specific decisions, such as selecting an appropriate package for the linear solver, need to be made, the process described in Section 4.4.2 is adapted. The nonfunctional goals are replaced by nonfunctional variable requirements and the available techniques are replaced by available choices to the specific decisions, such as available packages for the linear solver.

4.6 Summary

This section lists the names of all the common and variable requirements for FFEMP and discusses the scope of GOCA.

4.6.1 Common and Variable Requirements for FFEMP

All Common Requirements (CR) for FFEMP are listed in Figure 4.2 and all Variable Requirements (VR) for FFEMP are listed in Figure 4.3. Only the names of the requirements are given. The detailed description for each requirements can be found in Common and Variable Requirement Specification for FFEMP (Yu, 2010a).

Functional CR	Nonfunctional CR
FCR_Displace	NCR_Reliability
FCR_SameElmShape	NCR_Usability
FCR_SameNodeNum	NCR_Reusability
FCR_SameBElmShape	NCR_Maintainability
FCR_SameBNodeNum	
FCR_FileInput	
FCR_FileOutput	

Figure 4.2: Common Requirements for FFEMP

4.6.2 Scope of GOCA

GOCA is a part of FASCS, which is a methodology for developing SC program families. However, GOCA can also be used in the Domain Engineering stage

Functional VR	Nonfunctional VR
FVR_ElmShape	NVR_Accuracy
FVR_NumNode	NVR_Precision
FVR_IntMethod	
FVR_NumIpts	
FVR_StressS	
FVR_StrainS	
FVR_NumBNode	
FVR_BIntMethod	
FVR_NumBIpts	
FVR_Stress	
FVR_Strain	

Figure 4.3: Variable Requirements for FFEMP

of other methodologies for developing program family, as long as the problems to be solved can be modeled by continuous mathematical equations.

As discussed in the beginning of Chapter 3, it is believed that with some modification, GOCA can be used for developing program families to solve other types of problems, such as mesh generation problems, which are not modeled as continuous equations. A goal oriented approach has been used to develop software requirements for a mesh generation program, PMGT (Smith and Yu, 2009; Yu, 2007). The idea for eliciting, analyzing and documenting requirements for PMGT is similar to GOCA. One difference is that there is no common and variable related issues for PMGT since PMGT is not a program family. Another difference is that for PMGT, there is only one kind of models, which refines goals. Requirements are derived from these models. If only one kind of the models is kept, it is very likely that GOCA can be adapted to elicit, analyze and document domain requirements for the mesh generation program family.

Chapter 5

Family Member Development Environment

The idea of an environment for generating family members is presented by Weiss and Lai (1999). The use of the environment can significantly improve reusability, since it provides convenient facilities for application engineers to develop a family member. The process proposed by Weiss and Lai (1999) is general and they do not give a clear description of how to develop the environment. The proposed methodology, FASCS, is specific for SC software; therefore, more detail can be provided. This chapter is dedicated to the techniques used for developing a Family Member Development Environment (FMDE) for FASCS.

An FMDE is a set of tools that are developed in the domain engineering phase and are used by application engineers in the application engineering phase. As introduced in Section 3.1.2.2, a FMDE includes:

- a Domain Model (DM)
- a Variable Code Generator (VCG)
- a Test Case Generator (TCG)
- a Family Member Assembler (FMA)
- a Family Member Generation Process (FMGP)

The relationship between the tools in an FMDE and the stages of the program family development is shown in Figure 5.1.

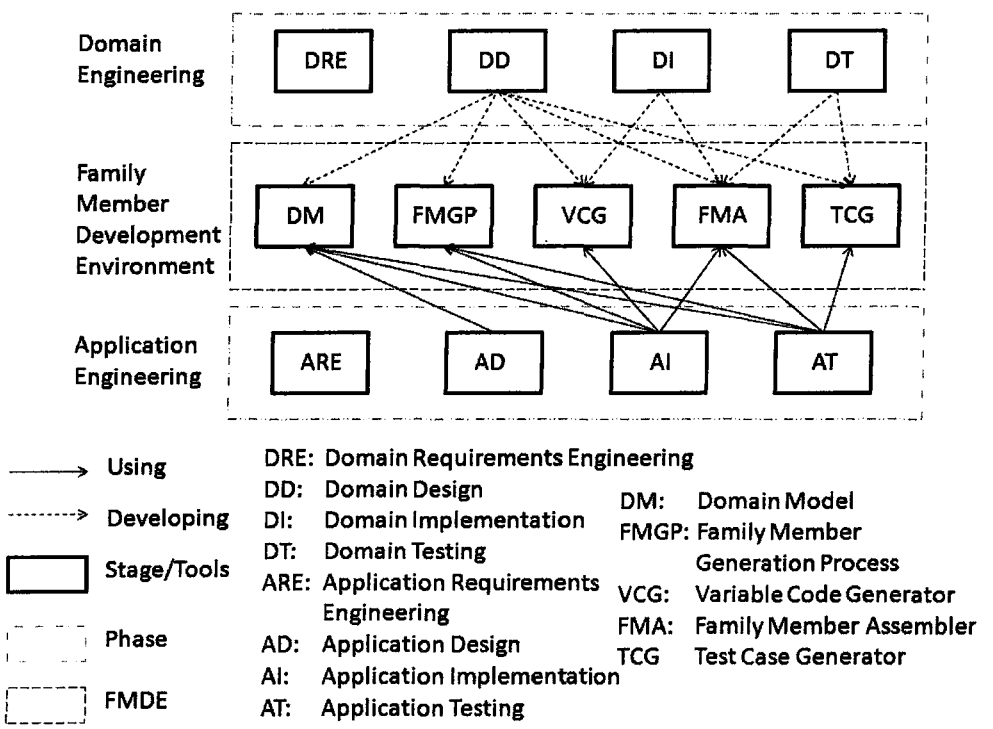

Figure 5.1: Relationship between Tools in FMDE and Stages of FFEMP

The development of an FMDE commences in the domain design stage. The DM and FMGP are developed in the domain design stage and the VCG, FMA and TCG are designed in the domain design stage. The VCG is implemented in the domain implementation stage and the TCG is implemented in the domain testing stage. The FMA needs to be implemented in the both domain implementation and domain testing stages.

When application engineers use the FMDE to generate family members, they use the VCG to automatically generate variable code and they use the TCG to automatically generate the variable part of some benchmark test cases. However, the use of both generators is implicit. Application engineers directly work with the FMA, which uses the VCG and TCG. The details on the use of the environment is specified in the FMGP. A specific family member is specified by the DM, which defines a language to describe the family members.

As described above, FASCS suggests using the automatic approaches to generating variable code and test cases and to assembling components for a specific family member. However, the FMDE can also be used for manually generating variable code or test cases or manually assembling components. In the former case, the VCG or the TCG is not included in FMDE and in the latter case, the FMA is not included. For example, the FMDE in the program family shown in Yu and Smith (2009), which was developed in the draft version of FASCS, included a VCG, but it did not include a TCG and an FMA.

The above components (tools) of the FMDE are discussed in the rest of this chapter. The example program family FFEMP, which was introduced in Section 2.4, is used for illustration purposes when the environment is specified.

5.1 Domain Model

A Domain Model (DM) can describe all possible members of the program family by specifying the variabilities of the members. The domain model for FASCS is similar to that for a general program family development process, as introduced in Section 1.2.1.3. A language, usually a Domain Specific Language (DSL), that specifies variabilities of the family is the key to a DM.

For some SC program families the type of possible values for the variabilities are simple, so a declarative language that is composed of a set of assignment statements is enough for specifying variabilities of most SC program families. The language defined by the DM for FFEMP, which uses BNF notations, is shown in Figure 5.2. An example family member of FFEMP,

```
statement::='ElmShape = 'shape|'NumNode = 'number|
            'IntMethod = 'imethod|'NumIpts = 'number|
            'StressS = 'state|'StrainS = 'state|
            'NumBNode = 'number|'BIntMethod = 'imethod|
            'NumBIpts = 'number|'Stress = 'bool|
            'Strain = 'bool
shape::= 'LINE'|'TRI'|'QUAD'|'TET'|'HEX'
number::=['1'-'9']+['0'-'9']*
imethod::= 'GAUSSQ'|'DIR'
state::= '<'bool','bool','bool','bool','bool','bool'>'
bool::= 'T'|'F'
```

Figure 5.2: The Definition of the Language for the Domain Model in FFEMP

whose variabilities were shown in Figure 3.2, is specified in Figure 5.3 by using the DSL that is defined in Figure 5.2.

Unlike those in Figure 5.2, variabilities for some SC program family are complicated. For example, the types of variabilities for the program family

ElmShape = TRI
NumNode = 3
IntMethod = GAUSSQ
NumIpts = 4
NumBNode = 2
BIntmethod = GAUSSQ
NumBIpts = 2
StressS = [F,F,T,F,T,T]
StrainS = [F,F,F,F,T,T]
Stress = F
Strain = F

Figure 5.3: The DSL Definition for the Family Member with Variabilities in Figure 3.2

described by McCuchan (2007) are mathematical equations, with types that represent functions, such as $\mathbb{R}^6 \times \mathbb{R} \longrightarrow \mathbb{R}$. The languages for the domain models representing this kind of variability are more complicated. For example, the modeling language for McCuchan (2007) is a subset of Maple (Maplesoft, Retrieved 2010).

5.2 Variable Code Generator

A Variable Code Generator (VCG) can automatically generate variable code. The VCG for FASCS is similar to what is implicit in Weiss and Lai (1999). For most of the SC application domains, automatically generating variable code has more benefits than using a traditional approach, because the generation of variable code for SC program families usually requires large amounts of calculation, even though the generated variable code may be small. The automatic approach improves reusability and addresses the modification challenge

introduced in Section 2.2.1, since it can help application engineers quickly developing specific family members. As discussed in Section 2.2.2, these application engineers are usually professional end users who develop and use the specific family members. The FASCS process does not assume that they have a background in software engineering. The use of VCG can significantly save them time and effort.

VCG can automatically generate family members that might not be produced using the traditional approach. For example, VCG can generate a member of FFEMP that solves very complication boundary conditions, such as tractions represented by polynomials of degree 5. Since this family member is not usually used for solving practical problems and the generation of this member needs a large amount of calculation, this member is usually not implemented without VCG. However, this kind of member can be used by scientists to thoroughly test their theories and gain more confidence for the correctness of their theories, as Section 2.1 discussed. Moreover, this kind of member can also be used to validate the design of the program family since the generation is easy. Thus, the reliability of the family is improved.

Among the technologies for building generators, a simple stand-alone program is used for VCG since the computation is a major concern for SC program families. Other technologies, such as using built-in metaprogramming capabilities and a generator infrastructure, will not be discussed further in this thesis. Details on these technologies can be found in Czarnecki and Eisenecker (2000).

Since macros are used for implementing the common code for FFEMP,

the generated variable code can be stored in a header file. An example generated file using the VCG of FFEMP for the family member defined in Figure 5.3 is partially shown in Figure 5.4. The first several lines define constants that are directly copied from the variabilities. The constants NUM_S, NV and NNE are simple constants that are calculated from variabilities. NUM_S indicates the size of constitutive matrix. NV represents the number of vertices for an element. NNE is the number of nodes in an edge. The constant SHAPEF defines the shape function of the element, which is $\{s, t, 1-s-t\}$ for a three node triangular element. The constant DNDSF defines the derivative of the shape function with respect to s, which is $\{1, 0, -1\}$. Both expressions are simplified in this thesis, for display purpose, by removing unnecessary zeros. How the complicated constants, such as SHAPEF, are calculated can be found in Appendix A. Since the generated code is lengthy, not all of it is displayed.

The programming language used to implement VCG depends on the characteristics of the program family. FFEMP use Matlab (Mathwork, Last Access 2010) to implement VCG because variable code for FFEMP needs to compute derivatives and integrations of mathematical equations. Matlab has a symbolic toolbox that can symbolically compute the derivatives and integrals.

The use of VCG does not conflict with the reuse of existing programs or components of the programs. In fact, FASCS strongly recommends reuse of existing programs or components. The Family Member Assembler that will be discussed in Section 5.3 is designed to automatically assemble the various components of a family member.

```
#define DIM 2
#define DOF 2
#define SHAPE TRI
#define NUM_NODE 3
#define INT_METHOD GAUSSQ
#define NUM_INT_PTS 4
#define NUM_B_NODE 2
#define B_INT_METHOD GAUSSQ
#define NUM_B_IPS 2
#define STRESS_S [F,F,T,F,T,T]
#define STRAIN_S [F,F,F,F,T,T]
#define STRESS 0
#define STRAIN_S 0

#define NUM_S 3
#define NV 3
#define NNE 2

#define SHAPEF {(0.0)*pow(s,0)*pow(t,0)+\
     (1.0)*pow(s,1)*pow(t,0)+(0.0)*pow(s,0)*pow(t,1),\
     (0.0)*pow(s,0)*pow(t,0)+(0.0)*pow(s,1)*pow(t,0)+\
     (1.0)*pow(s,0)*pow(t,1),(1.0)*pow(s,0)*pow(t,0)+\
     (-1.0)*pow(s,1)*pow(t,0)+(-1.0)*pow(s,0)*pow(t,1)}

#define DNDSF {(1.0)*pow(s,0)*pow(t,0)+\
     (0.0)*pow(s,1)*pow(t,0)+(0.0)*pow(s,0)*pow(t,1),\
     (0.0)*pow(s,0)*pow(t,0)+(0.0)*pow(s,1)*pow(t,0)+\
     (0.0)*pow(s,0)*pow(t,1), (-1.0)*pow(s,0)*pow(t,0)+\
     (0.0)*pow(s,1)*pow(t,0)+(0.0)*pow(s,0)*pow(t,1)}

                    ⋮
```

Figure 5.4: Generated Constants for a Family Member Defined by Figure 5.3

5.3 Test Case Generator

A Test Case Generator (TCG) allows application engineers to generate some benchmark test cases by giving the values of the variabilities of the family member and some additional information, which depends on the characteristics of the program family. Since the amount of test data for SC software is usually very large, automatic generation of test cases is suggested for FASCS to improve reusability and address the input and output challenge that defined in Section 2.2.1. More discussions for automatically generating test cases for SC program families can be found in Chapter 6.

The Test Case Generator (TCG) can automatically generate the variable portions of the benchmark test cases. The common portion can be developed using a traditional approach. Techniques for VCG, which are discussed in Section 5.2, can be used for developing TCG, since the development of TCG should be easier than VCG. The output of TCG is testing data, which should be simpler than the mathematical expressions that are output from VCG.

The common part of test cases for FFEMP is developed using macros, which is the same technique used for developing common code. By using TCG for FFEMP, the generated header file for testing the family member given in Figure 2.1 is partially displayed in Figure 5.5. There are 9 nodes and 8 elements. The prescribe displacement $\bar{u} = 0\,m$ and the traction $\bar{t} = 1\,N/m^2$. Only the test data for the testing mesh is given, to keep the presentation concise. If the computational domain and boundary conditions get complicated, the testing code can simply be changed to reading test data from files, instead of being given in a header file. In this case, the efficiency of performing the

```
 // -----testing mesh -----
 // coordinates
#define TMESHCOORD {{0.0,0.0},{5.0,0.0},{10.0,0.0},\
{0.0,5.0},{5.0,5.0},{10.0,5.0},\
{0.0,10.0},{5.0,10.0},{10.0,10.0}}

 // total number of nodes and elements
#define TMESHTNODE 9
#define TMESHTELM 8

 // nodal connectivity for elements
#define TMESHNC {{0,1,4},{0,4,3},{1,2,5},\
{1,5,4},{3,4,7},{3,7,6},{4,5,8},{4,8,7}}

 // total number of boundary elements
#define TMESHTTBC 2
#define TMESHTDBC 4

 // traction nodal connectivity,  related element,
 // constraints and values
#define TMESHTBC {{2,5},{5,8}}
#define TMESHTBEE {2,6}
#define TMESHTBCON {{{1,0},{1,0}},{{1,0},{1,0}}}
#define TMESHTBCV {{{1,0},{1,0}},{{1,0},{1,0}}}

 // displacement nodal connectivity, related element,
 // constraints and values
#define TMESHDBC {{0,1},{1,2},{0,3},{3,6}}
#define TMESHDBEE {0,2,1,5}
#define TMESHDBCON {{{1,1},{0,1}},{{0,1},{0,1}},\
{{1,1},{1,0}},{{1,0},{1,0}}}
#define TMESHDBCV {{{0,0},{0,0}},{{0,0},{0,0}},\
{{0,0},{0,0}},{{0,0},{0,0}}}

                    ⋮
```

Figure 5.5: Generated Constants for Testing a Member as Shown in Figure 2.1

tests would be decreased.

5.4 Family Member Assembler

Similar to what is implicit in Weiss and Lai (1999), the Family Member Assembler (FMA) can automatically assemble different components for a specific family member. The components include the common and variable components. The common components can be code developed by domain engineers and the variable components can be code generated by VCG. Both common and variable components can also be existing ones that were developed by others. The components may be a piece of code, libraries, or binaries. The FMA should have the ability to choose appropriate components and configure and assemble them together. As for using a VCG, using FMA can improve reusability since generating a family member becomes easier.

The development of FMA relates to the VCG. Sometimes, they are not separate and are developed together for programs that automatically generate both common and variable code (Carette, 2006; Elsheikh, 2010). However, end user developed SC software often needs to be modified. Sometimes, evolving the whole program family cannot be avoided. If FMA and VCG are developed as a whole, it has to be changed when the program family needs to evolve. Modifying a code generator may be nontrivial, especially for professional end users, who have little software engineering background. If FMA and VCG are developed separately, only common code needs to be modified when the program family needs to evolve. Hence, separating VCG and FMA can improve

the reusability of a program family.

Different than Weiss and Lai (1999), the FMA in FASCS also has the ability to assemble common and variable portions of some benchmark test cases. The approach of assembling test code is similar to assembling code for the family member.

For FFEMP, the major job of FMA, which is a Matlab program, is validating the family member specified in the language that is defined in Section 5.1 using constraints given in the Common and Variable Requirement Specification (Yu, 2010a). The FMA also calls the VCG to generate proper header files that define the variable code as constants in the directory of source files. If the test cases are required, the FMA calls the TCG to generate proper header files that define the test cases for the family member in the directory of testing files.

5.5 Family Member Generation Process

The Family Member Generation Process (FMGP) specifies how to use the DM, VCG, TCG and FMA to generate a specific family member. For example, the steps of generating family member for FFEMP are shown below. In this example, there is no external component for the common and variable components of the program family and the automatic approach is used for generating variable code and test cases and for assembling the components.

1. Determine the values of variable requirements for the family member.

2. Design the interfaces that are not completely specified in the domain

engineering phase.

3. Describe the family member by specifying variabilities using the Domain Specific Language shown in Table 5.2.

4. Specify the benchmark test cases by giving the computational domain, the way that the domain is meshed, the material properties and the boundary conditions.

5. Use the FMA to assemble the common code that is developed in the Domain Engineering phase and variable code that is automatically generated using VCG.

6. Use the FMA to assemble the common and variable part of testing code. The common part is developed in the Domain Engineering phase and the variable part is automatically generated using TCG.

Chapter 6

Implementation and Testing

This chapter discusses the implementation and testing of an SC program family using FASCS. In addition, a new testing technique to target the unknown solution challenge of SC software is proposed.

6.1 Implementation

When using FASCS to implement a program family, common code is developed in the domain implementation stage and variable code is developed in the application implementation stage.

There are a variety of techniques to develop reusable code (Czarnecki and Eisenecker, 2000). Generic programming is recommended, since the structure of SC programs is usually simple and the interactions between end users and the programs are limited. The only interaction for most SC programs is that the end users call the program and then receive the calculated results. If it is necessary, components-based programming (Heineman and Councill,

2001), which is more complicated than generic programming, can be used for more involved program family structure.

Another technique that is used by many program family developers is code generation. Code generation can be considered to avoid the performance penalty of generic programming, but this approach is more complicated. Professional end user developed SC programs, often required frequent changes. Modifying a code generator is nontrivial especially for end user developers who usually lack of software engineering background.

In addition, Aspect-Oriented Programming (AOP), which was discussed in Section 2.3.4.3, is also mentioned in Czarnecki and Eisenecker (2000) as technique for developing reusable code. However, AOP has limitations for FASCS because there are not many cross-cutting aspects in SC programs.

Since a Family Member Development Environment (FMDE) is used, the implementation of the variable part of the program family is simple. The variable code can automatically be generated using the Variable Code Generator (VCG), which is a component of FMDE. What the application engineers need to do is provide a program that contains variabilities for the specific family member. This program is written in a Domain Specific Language (DSL) that was developed in the Domain Design stage as a part of FMDE. The code for the family member consists of the common and variable code that are assembled together using the Family Member Assembler, which is also a component of the FMDE.

FFEMP uses macros, which is one of the simplest forms for generic programming to implement the program family. The variable code for FFEMP is

a set of mathematical expressions that are defined as constants. Since FFEMP uses C++, these constants can be defined in a header file. Therefore, the body for the code of the functions that may vary for some family members is the same. The differences are the values of constants. The reason for the simplicity of the implementation is that the family is designed such that the complicated computation is performed by the VCG. The generated variable code itself is relatively simple. The example of generated header file was shown in Section 5.2. The computation of one of the generated constants, SHAPEF, for FFEMP is illustrated in Appendix A.

The characteristics of the program family and the nonfunctional requirements should be considered when selecting the programming language for the implementation. In addition, the programming languages that other similar programs use should be considered. Although the application engineers do not need to write any code when using VCG and FMA, it is possible that the common code needs to be changed when the whole program family evolves. The use of programming languages that are widely used in the community of the domain can improve reusability. For example, there is more than one programming language that could be used for FFEMP, but C++ was selected because many Finite Element Method (FEM) programs are developed using C++. Modification occurs frequently in the reuse of FEM software. Using C++ makes it possible for some end user developers of FEM software to consider investigating FFEMP.

If it is possible, FASCS suggests reuse of existing code, libraries or components. However, there are pros and cons for the reuse, especially for

the reuse of libraries. The installation of required packages may decrease the usability because some libraries may not be well documented and users may not have knowledge about the architecture of their computer. For example, FFEMP needs to solve sparse linear system. There are many existing libraries, such as LAPACK (Anderson et al., 22 Aug 1999), that could fulfill this task. However, installing LAPACK is still not trivial for some users, although the documentation for LAPACK is well developed. Rather than having users struggle with libraries, an existing piece of code, CSparse (Davis, Last Access 2010), is used. CSparse is "inserted" into FFEMP code; thus, the installation of a linear solver library is eliminated.

6.2 Testing

The purpose of the proposed methodology, FASCS, is to improve the quality of SC software. Testing is an important approach to measuring quality. In addition, testing can improve some quality factors, such as reliability. This section focuses on dynamic testing.

As mentioned in Section 5.3, the automatic generation of test cases can improve reusability. However, test cases for members of SC program families are apparently never generated automatically, although the automatic generation of test cases has been applied in many applications (Perrouin et al., 2010; Uzuncaova et al., 2010).

The test cases to be automated are the benchmark test cases mentioned in Section 3.1.4. These benchmark test cases are designed manually.

For example, the shape of the computational domain and the type of boundary conditions for the benchmark test cases of FFEMP are determined and cannot be changed. The magnitude of the domain and boundary conditions, the number of elements and all variabilities are the input to the Test Case Generator and can be changed.

As for implementation described in Section 6.1, the generic programming can also be used to develop test cases. The common part of test cases is developed in the domain testing stage. The variable part, which is assembled with the common part to form a set of complete test cases, is developed in the application testing stage.

The testing for routines that are common for all family members is similar to testing routines in a single program. However, dynamically testing common code in the routines that vary for some family members is difficult because the variable part of the code is missing and the routine cannot be executed until the missing variable part of the routine is developed in the application implementation stage. The test cases for this kind of routine are often parameterized (McGregor, 2001). That is, the dynamic tests for common code in the variable routines cannot be performed in the domain testing stage, which may increase the cost of the development because the earlier the test is conducted, the less cost is required to fix the defect.

This drawback can partially be overcome by using FASCS. Since variable code can automatically be generated by using a Variable Code Generator, domain engineers can generate missed variable code for the routines that vary for some family members. Also, the Test Case Generator (TCG), which is a

part of the Family Member Development Environment (FMDE) that was discussed in Chapter 5, can generate variable part of some benchmark test cases for testing the common part of the variable routine. Since these test cases are designed for testing common code in variable routines, the test cases are usually not thorough for testing variable code. For example, it is not feasible for the TCG in FFEMP to generate all kinds of triangular elements to test their shape functions. In addition, another set of benchmark test cases used for testing nonfunctional requirements, such as accuracy and reliability, can also be generated by TCG. This approach can also be used for the integration testing in the domain testing stage.

A challenge for the above approach is the large number of tests that need to be conducted when the number of variabilities that impact the variable routine is increased. This is also a challenge for the integration testing that all program family testing faces. A program family has numerous possible members, even with only a few variabilities. For example, there are 11 variabilities for FFEMP. The number of variants for each variability is listed in Table 6.1. Let n_f represent the total number of family members. Then n_f can be calculate by

$$n_f = 5 * \text{MN} * 2 * \text{MIP} * 2^6 * 2^6 * \text{MBN} * 2 * \text{MBIP} * 2 * 2$$

If the constants for the maximum values of variabilities are assigned as follows: MN = 28 (the max number of nodes), MIP = 16 (the max number of integration points), MBN = 8 (the max number of boundary nodes)

Variability	Number of Variants
FVR_ElmShape	5
FVR_NumNode	MN
FVR_IntMethod	2
FVR_NumIpts	MIP
FVR_StressS	2^6
FVR_StrainS	2^6
FVR_NumBNode	MBN
FVR_BIntMethod	2
FVR_NumBIpts	MBIP
FVR_Stress	2
FVR_Strain	2

Table 6.1: The Number of Variants for Variabilities of FFEMP

and MBIP = 5 (the max number of boundary integration points), then $n_f = 5,872,025,600$, which means that there are $5,872,025,600$ combinations. Assume that FVR_Stress = $\langle F, F, T, F, T, T\rangle$ and FVR_Strain = $\langle F, F, F, F, F, F\rangle$, which means that FFEMP can only solve for plane stress problem and the number of variants for FVR_StressS and FVR_StrainS are 1. With some constraints between the variabilities applied, such as those that were shown in Section 4.5.2, the number of possible family members can be reduced. For example, the constraint RC_mi in Equation 4.5 shows that the FVR_NumIpts is a requirement for a member of FFEMP if and only if FVR_IntMethod = GAUSSQ. This constraint can reduce the combination of the number of variants for FVR_IntMethod and FVR_NumIpts from $2*16 = 32$ to $1+16 = 17$. However, there are still $63,648$ possible family members by applying all constraints for FFEMP, which are given in the Yu (2010a). It is difficult, if not impossible, to develop all of the possible family members and to test all the combinations of

these variabilities. Hence some techniques have to be used to reduce the number of combinations. McGregor (2001) suggests two techniques to mitigate this problem: combinatorial test designs and incrementally integration testing. These two techniques are options for designing and conducting integration tests using FASCS. These techniques are not discussed in this thesis and more details can be found in Burr and Young (1998) and McGregor (2001).

Another challenge that is specific for SC software is the unknown solution challenge as discussed in Section 2.2.1. A new testing technique, named Computational Variability Test, is proposed to contribute to resolving this challenge. The details of this technique are given in the next section.

6.3 Computational Variability Test

Computational Variability Test (CVT) is a testing technique that is used to test common code in variable routines. More frequently, it is used to perform integration test for SC program families. No analytical solutions are necessary for using this technique. The prerequisite for using CVT is that the variable routines or the program family to be tested vary on at least one computational variability.

A computational variability is a variability that only relates to the computation techniques used to solve the theoretical model of the program family. The computational variabilities do not impact the theoretical model. For example, among the 11 variabilities of FFEMP that are listed in Table 4.3, 7 of them (FVR_ElmShape, FVR_NumNode, FVR_IntMethod, FVR_NumIpts,

FVR_NumBNode, FVR_BIntMethod and FVR_NumBIpts) are related to FEM, which is the computational technique used to solve the theoretical model (Equation 2.1). Hence, they are computational variabilities. Another four variabilities (FVR_StressS, FVR_StrainS, FVR_Stree and FVR_Strain) are not.

The steps of using CVT to test variable routines or a program family are listed below. It is assumed that at least one of variabilities is a computational variability. It is also assumed that the computational variabilities for the program family are identified and input of the routines or a program family that is not related to the computational variabilities are given .

1. Select the appropriate possible values for each computational variability.
2. For each possible value of each computational variability:
   (a) Specify the member of the program family with the corresponding variabilities using the Domain Specific Language developed for the program family.
   (b) Use the Variable Code Generator to generate variable code for the family member.
   (c) Use the Test Case Generator to generate test cases for the family member.
   (d) Use the Family Member Assembler to assemble the code for the family member and the test case to test the family member.
   (e) Execute the test case and record the test results.
3. Compare the test results for the family members.

The test is considered to pass if the pairwise relative differences of the test results are within an acceptable range. This acceptable range depends on the possible values selected in Step 1 and the requirements of accuracy and precision of the program family. For the testing to be meaningful, the possible values selected in Step 1 should not be too extreme such that the difference between the solution to the theoretical model and the solution to the computational model is not too big. For example, if the computational domain for FFEMP had a curved boundary, selecting a mesh with a few elements or selecting a small number of nodes per element in Step 1 would not be proper.

The test result for the example problem shown in Figure 2.1 is shown in Table 6.2. The length of the domain $L = 9\,m$ and the width $W = 6\,m$. The

Number of Node	3	3	6
	6	10	10
Relative Differences	1.1111×10^{-12}	5.0927×10^{-12}	4.8149×10^{-12}

Table 6.2: The Relative Difference for Test Case shown in Figure 2.1

domain has unit thickness ($H = 1\,m$). The Young's modulus $E = 1000\,N/m^2$ and the Poisson's ratio $\nu = 0.3$. The traction $\bar{u} = 1\,N/m$. The computational domain is meshed to Figure 2.2. The computational variability is the number of nodes per element. The maximum relative difference is 5.0927×10^{-12}, which is acceptable for FFEMP. Hence, this test passed.

CVT can also be used to check the sensitivity of the theoretical models. If the calculated result is sensitive to the change of a specific computational variability, then the theory under development may be sensitive to the assumptions related to the computational variability.

Although the closed form solutions are not necessary, the test to a problem with a closed form solution can also use CVT. The closed form solutions for displacements u and v to the test case shown in Figure 2.1 can be calculated as:

$$\sigma_x = \frac{F_x}{A_x} = \frac{\bar{u} * W}{W * H} = 1\, N/m^2$$

$$\epsilon_x = \frac{\sigma_x}{E} = 1/1000 = 0.001$$

$$\epsilon_x = \frac{du}{dx} \Longrightarrow u = \int_0^x \epsilon_x\, dx = 0.001 * x$$

$$\epsilon_y = -\epsilon_x * \nu = -0.0003$$

$$\epsilon_y = \frac{dv}{dy} \Longrightarrow v = \int_0^y \epsilon_y\, dy = -0.0003 * y$$

The calculated displacements with different number of nodes per element can be obtained for meshing the computational domain as shown in Figure 2.2. The relative errors of the calculated results shown in Table 6.3. Since both

Number of Node	3	6	10
Relative Error	1.5462×10^{-13}	4.8181×10^{-13}	3.1664×10^{-12}

Table 6.3: The Relative Error for Test Case shown in Figure 2.1

displacements are linear, using 3 nodes per element can obtain exact solution. The more the number of nodes per element used, the more calculation needed. Hence, the bigger the relative error shown in Table 6.3.

A crucial step to efficiently and effectively use CVT is the selection of the appropriate values for the computational variabilities. It is not possible to choose all possible values for all computational variabilities, as the previous

section discussed. On the other hand, the test results of too few values for some computational variabilities would not add much confidence on the accuracy and precision of the program family.

Selecting a key computational variability is the solution to the above problem. A key computational variability is one that impacts the calculation much more than the other computational variabilities. For example, the number of nodes per element (FVR_NumNode) is a key computational variability for FFEMP. The selection of a key computational variability also depends on the purpose of testing. For example, if one did not feel confident in the code related to integration, the number of integration points (FVR_NumIpts) might be selected as the key computational variability.

To balance the efficiency and effectiveness of CVT, the number of values for these key computational variabilities can be selected more than other computational variabilities, so that the number of detected defects is maximized without the need to generate too many family members.

The reason for CVT being possible is the existence of the Variable Code Generator, Test Case Generator and Family Member Assembler, so that the generation of a family member and the test cases is easy. Otherwise, CVT is too inefficient for practical use. Moreover, if a manual approach is used it is very difficult to ensure that the assumed common code between family members is actually common. It is too easy to inadvertently modify the code in a subsequent family member.

CVT is a valuable testing technique because it allows for the comparison of quasi-independent solution algorithms. For programs with unknown

solution, one can build confidence by comparing different solutions that solve the same problem. If two independent solutions agree, this provides some verification for both programs. Even though two members of the same program family is not independent, because much of the code is still common, modifications that do not change the solution are a positive sign. Changing computational variabilities can be used in a manner analogous to grid refinement studies (Roache, 1998), as Section 8.2 will discuss.

Chapter 7

Documentation

Documentation is a very important part of a program family. A well documented program family should be easy to understand and thus easily used by application engineers to generate members of the program family. Moreover, documentation improves reliability and maintainability, as Yu (2007) and Smith and Yu (2009) discussed.

This chapter presents documents for FASCS, which include documents for domain requirements engineering (Section 7.1) and domain design (Section 7.2). The relationship between documents and stages of the program family development is shown in Figure 7.1. In addition, a traceability matrix between artifacts of the domain requirement engineering stage and that of the domain design stage are given in Section 7.3. Traceability matrices between artifacts within the domain requirement engineering stage are developed in Common and Variable Requirement Specification (Section 7.1.1) and traceability matrices between artifacts within the domain design stages are developed in Ref-

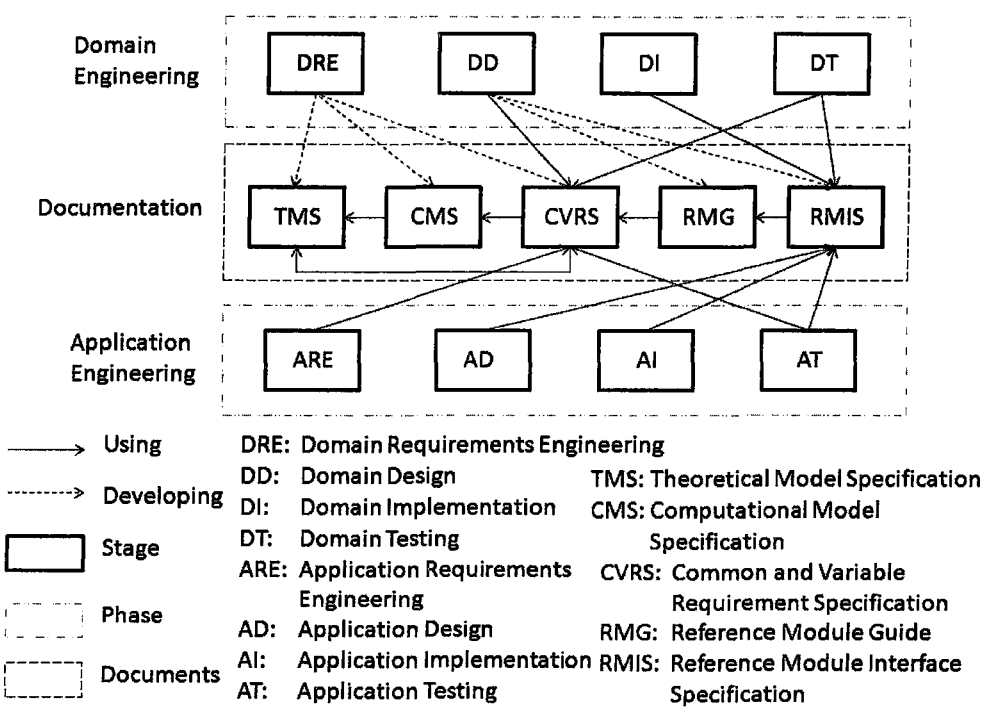

Figure 7.1: Documents

erence Module Guide (Section 7.2.1). Again, the proof of concept program family, FFEMP, is used as an example when the documents are discussed. It is assumed that the readers of this chapter have read the previous chapters and understand the overall process.

Since some content in one document can be reused in other documents, tools should be used to keep the reusable portion consistent between documents, to improve the reusability and maintainability. For example, equations that represent theoretical models for FFEMP appear in the Common and Variable Requirements Specification, Theoretical Model Specification and Computational Specification. These equations are documented in a separate file. Since all documents for FFEMP are written in Latex (LaTeX, Retrieved July 2010), it is possible that these equations can be "input" to the locations where they are required. If these equations need to change, only the file that stores the equations are changed. The documents that embed the equations are updated automatically.

There is a distinction between sections in this chapter and sections in the documents for a program family. Numbers of the sections are used when the sections of this chapter are referenced, such as Section 7.1.1.3. Names of the sections are used when the sections refer to the documents for a program family, such as the *Common and Variable Requirements* section.

The template for each document is given in the beginning of the description. Theses templates are inspired by Chen (2003); Lai (2004); Yu (2007); Yu and Smith (2009). Modifications to previous templates are necessary because they do not entirely meet the current use. For instance, Chen (2003) only

gives the template for documenting requirements without much explanation or rationale for it. Lai (2004) and Yu (2007) use a goal-oriented approach. However, the templates they proposed are not for a program family. Yu and Smith (2009) do not separate the documents for the theoretical model and computational model.

The process of developing documents is iterative. The order of documents for stages means that the development of the later stage requires the documents for previous stage in each iteration. The order also gives the direction of reading the final document. That is, readers can read the documents as if the documents are developed in the order that they appears. This is what Parnas and Clements (1986) proposed for their rational design process.

In addition to the documents for all stages of the process, the portions in each document are also developed iteratively. The order inside each document provides the prerequisite of the portions for each iteration. Similarly, the documents for each stage can be read as if the documents were developed in the order that they appear.

The specification for the requirements and design use formal mathematical expressions to avoid ambiguity. The unambiguous requirements and design make the assessment of accuracy, precision and reliability possible. All mathematical expressions in FASCS follow the format given by Hoffman et al. (1995).

When applicable, a prefix is used to number and label items, such as goals, requirements, models, *etc.* The prefixes and the corresponding represented items are listed in the Table 7.1. A convention that the prefix of the

label is the prefix of the number followed by an underscore "_" is used in the documents for a program family.

FG	Functional Goal
NG	Nonfunctional Goal
FCR	Functional Common Requirement
NCR	Nonfunctional Common Requirement
FVR	Functional Variable Requirement
NVR	Nonfunctional Variable Requirement
TTD	Theoretical Terminology Definition
CTA	Common Theoretical Assumption
TM	Theoretical Model
VTA	Variable Assumption
CTD	Computational Terminology Definition
CCA	Common Computational Assumption
CM	Computational Model
AC	Anticipated Change
UC	Unlikely Change
M	Module
GC	General Constraint
RC	Requiring Constraint

Table 7.1: The Prefixes for Numbers and Labels

7.1 Domain Requirements Engineering

The major document for the domain requirements engineering stage is the document for common and variable requirements. Since FASCS uses GOCA to perform commonality analysis, the theoretical model and the computational model can be reused. Hence, it is convenient to document the theoretical model and the computational model separately.

Goals, including functional goals and nonfunctional goals, of the pro-

gram family are documented in the major document for GOCA: the Common and Variable Requirements Specification (CVRS). As its name suggests, the specification for common and variable requirements are the major part of CVRS. The theoretical model is documented in the Theoretical Model Specification (TMS). The computational model is documented in the Computational Model Specification (CMS).

To fully understand a program family, the TMS and CMS should be read before the CVRS, except for goals of the program family, which should be read first because they determine TMS and CMS. In addition, TMS should be read before CMS. Certainly, any of the documents is understandable if a reader already understands the essential part of the program family.

7.1.1 Common and Variable Requirements Specification

Common and variable requirements are specified in CVRS. The template of CVRS is shown in Figure 7.2. The details for each section are given below.

7.1.1.1 Reference Material

This section serves as a reference. It including symbols, abbreviations and acronyms, and auxiliary constants that are used in this document (CVRS) and the other two documents for the domain requirements engineering stage, TMS and CMS, since these three documents are strongly related. The purpose of the *Reference Material* section is to ease the use of the documents and to improve their understandability.

1 Reference Material
2 Introduction
  2.1 Purpose of the Document
  2.2 Introduction to Program Family
  2.3 Introduction to Domain Area
  2.4 Organization of the Document
3 Common and Variable Requirements
  3.1 Goals
  3.2 Determination of Variable Assumptions
  3.3 Theoretical Refinement
    3.3.1 Theoretical Model
    3.3.2 Refinement of Some Theoretical Terminology
  3.4 Computational Requirement
    3.3.1 Computational Model
    3.3.2. Refinement of Some Computational Terminology
  3.5 Common Requirements
  3.6 Variable Requirements
4 Other System Issues
  4.1 Open Issues
  4.2 Off-the-sell Solutions
  4.3 Waiting Rooms
5 Traceability Matrices

Figure 7.2: The Template for Commonality Analysis

7.1.1.2 Introduction

This section gives an overview of CVRS. The purpose of the document and the intended audience of this document is provided. Since the program family approach is relatively new to SC software developers, the process of the program family approach is also briefly introduced. In addition, a reference to detailed discussion of the program family is included. The background information for the domain is given because the user of the document may not be an expert on the domain area. The organization of the document is summarized at the end of the introduction section.

7.1.1.3 Common and Variable Requirements

This section is the major portion of the commonality analysis report. All common and variable requirements are developed in this section.

Goals

Both functional and nonfunctional goals are documented in this section. Since goals are allowed some degree of ambiguity, plain text is used for specifying goals. However, names and numbers should be given for further reference and traceability. Functional goals can be used in multiple documents. Hence, they should be stored in a separate file, as discussed in the beginning of this chapter. Nonfunctional goals are simple; numbers and names are enough to understand their meanings. Therefore, nonfunctional goals do not need to be stored in separate files. Examples of functional goals and nonfunctional goals for FFEMP are given in Section 4.1.

Determination of Variable Assumptions

The theoretical model is assumed to have been developed and documented in TMS (see Section 7.1.2), before the development of this section. This section determines and documents variable assumptions, which were not determined in the TMS. Some examples of variable assumption were given in Section 4.3.

Theoretical Refinement

First, the equations that represent the theoretical model, which has been developed and documented in TMS, is restated for quick reference. Second, some theoretical terminology is refined with respect to the determined variable assumptions, which are documented in the *Determination of Variable Assumption* section.

Taking FFEMP as an example, if the variable assumptions were determined as VTA_StressS = $\langle F, F, T, F, T, T\rangle$ and VTA_StrainS = $\langle F, F, F, F, F, F\rangle$, which means that the computational domain is 2D, the theoretical terminology *Stress* (TTD_Stress), which is defined in TMS as Equation 4.2 would be refined to Equation 4.3 in this section. Except for TTD_Stress and the other four theoretical terminology definitions: *displacement* (TTD_Displace), *strain* (TTD_Strain), *Linear Differential Operator* (TTD_LDOpt) and *Constitutive Matrix* (TTD_Constitutive), other parts of the documents for the Domain Requirement Engineering stage, which includes Common and Variable Requirement Specification, Theoretical Model Specification and Computational Model Specification, remain the same. This indicates that the documents for FFEMP are easy to reuse.

Computational Refinement

It is assumed that the computational model has been developed and documented in CMS, which will be discussed in Section 7.1.3, before the development of this section. The equations that represents the computational model are redisplayed first. Then, some computational terminology is refined with respect to the determined variable assumptions. However, for the FFEMP example, this case does not arise because there are no computational terminology definitions that need to be changed when the variable assumptions change for FFEMP.

Common Requirements

Both functional common requirements and nonfunctional common requirements are documented in a table format. This format has been successfully used in Chen (2003), Lai (2004), Yu (2007) and Smith et al. (2009). The table format is also used for documenting other portions, such as functional variable requirements, of CVRS and other documents, such as TMS, for the program family. There are five fields, namely Number, Label, Related Items, Description and History, that are used to document the common requirements. These fields are also frequently used in other program family documents.

Number: The number is used for purposes of cross-referencing and traceability within the same documents.

Label: The label is a short identifying phrase. This label provides a mnemonic that helps with quickly remembering which definition is being presented. Moreover, the label will be useful when an external document needs to

reference one of the definitions in this document.

Related Items: A related item is a item that is used by the current item. For example, a model, a terminology definition, an assumption or another requirement that is used by the current requirement is a related item for the description of a requirement. Should the related items change, the current requirement will also need to be modified.

Description: The actual specification is given here. In some cases where the description is lengthy, some of the details are moved to a section following the table.

History: A history includes the creation date and any subsequent modifications.

Example table format specifications of a functional common requirement and a nonfunctional common requirement for FFEMP were shown in Section 4.5.1 (Table 4.1 and Table 4.2).

Variable Requirements

Functional variable requirements are documented in a table format. In addition to Number, Label, Related Items, Description and History, which are the same for fields in the common requirement specification discussed above, the specification for a functional variable requirement has the following fields:

Type: When applicable, the type of the specified item is given. The type may be real ($\mathbb{R}$), integer ($\mathbb{N}$), boolean ($\mathbb{B}$), string ($\mathbb{S}$), or set. The type information helps to clarify the meaning of the variable requirement.

Dependency: Dependency specifies a relation between the functional variable requirement and a member of the program family. Dependency can be Mandatory or Optional, meaning that this functional variable requirement is meaningful for all or some member of the program family, as discussed in Section 4.5.2. If the functional variable requirement is optional, the conditions that a member of the program family has this requirement should be specified in the "Constraints" field.

Constraints: The name of any constraints between this functional variable requirement and other ones should be given. The constraints should be expressed as a boolean expression and the value of the expression should be true. The name of both the general and requiring constraints, whose prefixes are given in Table 7.1, are given in the table. The details of the constraints are given after of all functional variable requirements are specified.

Variants This field is used for the variable item that the table specifies. It provides a set of possible values for the items. The set of possible values is a subset of the "Type."

An example table format specification of a functional variable requirement for FFEMP is shown in Section 4.5.2 (Table 4.3).

A graphical notation for functional variable requirements is suggested. The representation of symbols and notations used in the graph is illustrated in Figure 7.3. This notation is a simplified from Kang et al. (1990) and Pohl et al. (2005). Since SC software usually has simple structure, not all of the no-

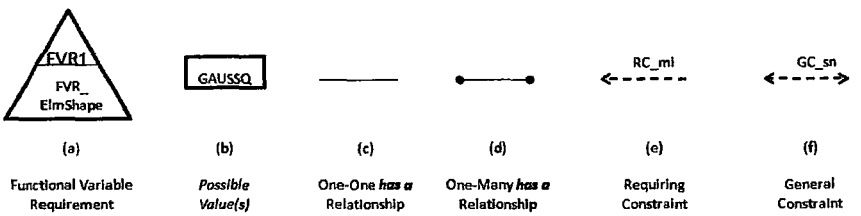

Figure 7.3: Symbols and Notations for the FVR Graph

tations in Kang et al. (1990) and Pohl et al. (2005) are used in the documents. In addition, a new notation that represents One-Many "has a" relationship, which is shown in Figure 7.3 (d), is proposed. The new notation serves the same purpose as the One-One "has a" relationship , but without the need to explicitly list all of the possible values. The new notation is proposed because some functional variable requirements for SC program family have relatively large numbers of variants and these variants are sets of integers. Because the excessive input data can be represented properly, readability of the graph is improved. The input output challenge for SC software mentioned in Section 2.2.1 is also addressed. It is also possible that the variants are set of real numbers, which is infinite. By slightly modification, the new notation can be used to denote this kind of variabilities, which cannot be represent graphically using the notations in Kang et al. (1990) and Pohl et al. (2005).

The graphical notations can improve the understandability of the family since it gives an overview of all functional variable requirements and their relations. These functional variable requirements can also help with understanding the scope of the program family.

An example of the functional variable requirements graphical notation for FFEM is shown in Figure 7.4. The figure clearly shows that there are eleven functional variable requirements. FVR_NumIpts and FVR_NumBItps are optional. Others are mandatory. The existence of FVR_NumIpts as a requirements for a member of FFEMP depends on the value of FVR_IntMethod (RC_mi) as discussed in Section 4.5.2. Similarly, FVR_NumBIpts is a requirement for a member of FFEMP if and only if FVR_BIntMethod = GAUSSQ. In addition, there are four general constraints, such as the constraint between FVR_ElmShape and FVR_NumNode (GC_ni) that was discussed in Section 4.5.2. The detailed specification on the general constraints can be found in the CVRS for FFEMP (Yu, 2010a).

Graphical notation for a program family, such as Figure 7.4, does not give all of the information of the functional variable requirements, such as the Related Items. It cannot fully replace the textual specifications. The textual specification of the Common and Variable Requirements for FFEMP is given in (Yu, 2010a).

Nonfunctional variable requirements are documented in a table format. This table contains Number, Label, Related Items, Variants and History. The type for all nonfunctional variable requirements is real number, as discussed in Section 4.5.2. In addition, all nonfunctional variable requirements are mandatory and constraints between nonfunctional variable constraints are difficult to specify quantitatively. Table 7.2 shows an example of the nonfunctional variable requirement for FFEMP.

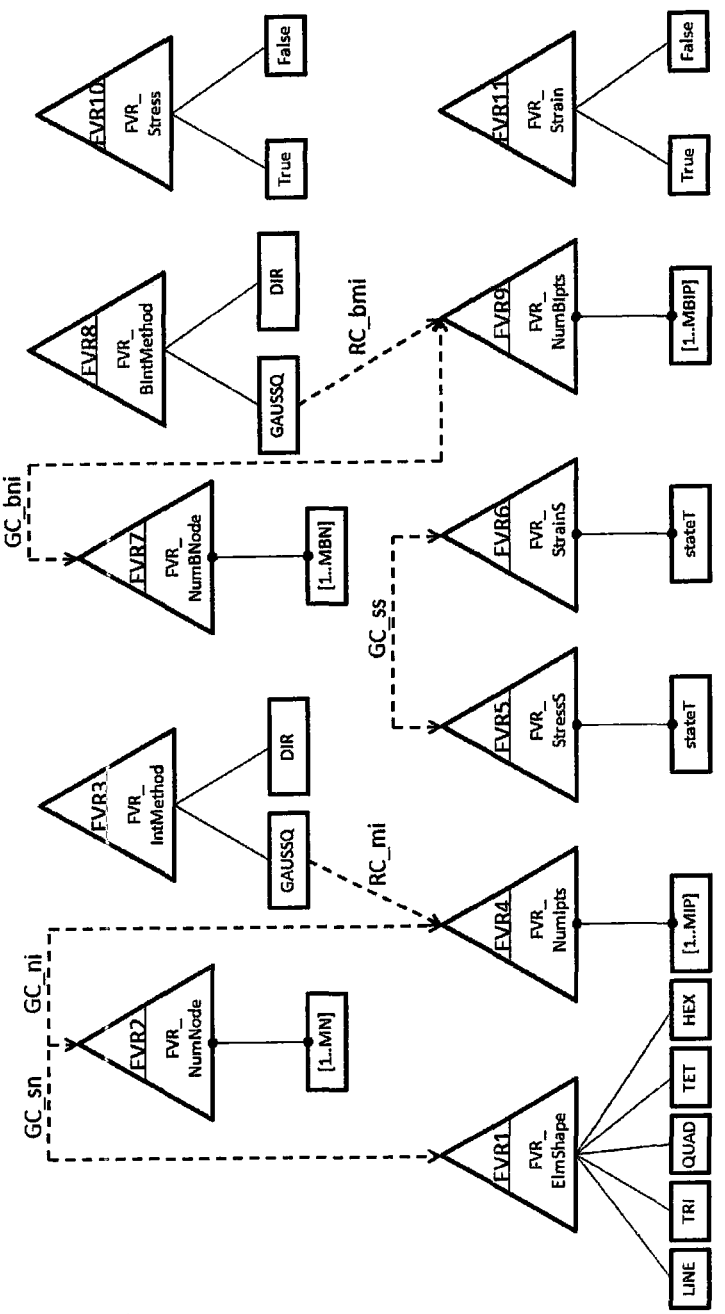

Figure 7.4: Graphic Notation of Functional Variable Requirements for FFEM

Number	**NVR1**
Label	**NVR_Accuracy**
Related Items	NG_Accuracy
Variants	{0.8333, 0.1667}
History	Created - January 2011

Table 7.2: An Example Nonfunctional Variable Requirement for FFEMP

7.1.1.4 Other System Issues

This section includes some other supporting information that might contribute to the success or failure of the system development. Open issues, off the shell solutions, and waiting room items are considered here. In particular, the waiting room items related to relaxing the common assumptions, including common theoretical assumptions and common computational assumptions, as discussed in Chapter 4. The change of variable assumptions is not included in the waiting room since the change only affect a small portion of changes for the program family, as Section 7.1.1.3 shows. An example of a potential relaxed assumption for FFEMP is that the thermal effects cannot be neglected, which is a change of common theoretical assumption CTA_Thermal. If this assumption were changed, the theoretical model would change. The waiting room provides a blueprint of how the system will be extended, and hence it improves the reusability and maintainability of the program family.

7.1.1.5 Traceability Matrix

The traceability matrix presented in this section includes the relations between artifacts in different documents, including CVRS, TMS and CMS. The

relations between most artifacts within CVRS, such as the relation between NG_Reliability and NCR_Reliability in FFEMP, are obvious. Hence only some relations that are not obvious are included in the matrix. The relations between artifacts within TMS and CMS are documented in the corresponding TMS and CMS.

The traceability matrix between terminology definitions and functional requirements for FFEMP is shown in Table 7.3. The check mark in an entry, such as (FCR_Displace, CTA_Displace) in Table 7.3, indicates FCR_Displace depends on CTA_Displace. That is, if CTA_Displace changes, then FCR_Displace will change.

	TTD_Poisson	TTD_Young	TTD_Constitutive	CTD_Mesh	CTD_Displace	CTD_Stress	CTD_Strain	CTD_Kinematic	CTD_Stiff	CTD_Load	FVR_IntMethod	CTD_BIntMethod
FCR_Displace	✓	✓		✓	✓				✓	✓		
FVR_NumIpts											✓	
FVR_StressS						✓						
FVR_StrainS							✓					
FVR_NumBIpts												✓
FVR_Stress			✓		✓	✓		✓				
FVR_Strain					✓		✓	✓				

Table 7.3: Traceability Matrix between Terminology Definitions and Functional Requirements for FFEMP

7.1.2 Theoretical Model Specification

Theoretical Model Specification (TMS) documents the theoretical model for a program family. The template of TMS is shown in Figure 7.5. The details on each section are presented below. It is assumed that the goals for the program family are identified before developing the TMS.

1 Introduction
2 Functional Goals
3 Theoretical Terminology Definitions
4 Common Theoretical Assumptions
5 Theoretical Model
6 Variable Assumptions

Figure 7.5: The Template for Theoretical Model Family Specification

7.1.2.1 Introduction

This section gives the purpose and the organization of the document.

7.1.2.2 Functional Goals

This section lists functional goals for quick reference, since the theoretical models are refined from functional goals. The restatement of functional goals makes the document relatively independent and easy to use.

7.1.2.3 Theoretical Terminology Definitions

This section explicitly defines theoretical terminology with respect to common theoretical assumptions, which are documented the *Common Theoretical Assumption* section of this document. The theoretical terminology definitions,

together with the computational terminology definitions that will be discussed in Section 7.1.3, should be documented as formally as possible to avoid ambiguity, since these definitions may be used for documenting common and variable requirements.

Theoretical terminology definitions are documented in table form. In addition to Number, Label, Related Items, Description and History, which are the same for fields in the common requirement specification discussed in Section 7.1.1, the specification for a theoretical terminology definition has the following fields:

Symbol: This field shows the symbol that is used to represent variables related to this concept. Tensors, matrices and vectors are represented by bold faced symbols, such as $\mathbf{u}$ or $\boldsymbol{\sigma}$.

Type: In addition to the type information given in Section 7.1.1, the type for a tensor, a matrix or a vector should also be specified. For a tensor, a matrix or a vector, the type specifies the type for each component. The type information helps to clarify the meaning of the symbol and the variables.

Units: Where applicable the units associated with the symbol are given. The unit system adopted is the "MLtT" dimension system, where M is the dimension of mass, L is length, t is time and T is temperature. This system corresponds nicely with the SI (Système International d'Unités), or modern metric, system, which uses units of kilogram (kg), meter (m), second (s) and Kelvin (K) for M, L, t and T, respectively. By leaving the

units in this form any unit system can be adopted in the future, as long as the choice of specific units is consistent between different quantities. When units are for tensor or vector quantities, it is implicit that the unit measure applies to the individual components.

Sources: This field lists references that can be consulted for additional information on the concept in question.

An example table format specification of a theoretical terminology definition for FFEMP is shown in Table 7.4, which defines the displacement. The

Number	**TTD1**
Label	**TTD_Displace**
Symbol:	**u**
Type:	$\mathbb{R}$
Units:	L
Related Items:	CTA5
Sources:	Bauld (1986)
Description:	Displacement of a point is defined as the distance between the point before and after the deformation.
History:	Created – July, 2010

Table 7.4: An Example Theoretical Terminology Definition for FFEMP

detailed description following the table can be found in the TMS for FFEMP (Yu, 2010b). Displacement can be represented by a vector. The general 3D displacement vector is given below, where u, v and w are in the x, y and z

directions, respectively.

$$\mathbf{u} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{7.1}$$

All theoretical terminology definitions for FFEMP are shown in Table 7.5.

Number	Label	Symbol
TTD1	TTD_Displace	$\mathbf{u}$
TTD2	TTD_Stress	$\boldsymbol{\sigma}$
TTD3	TTD_Strain	ϵ
TTD4	TTD_LDOpt	$\mathbf{L}$
TTD5	TTD_Poisson	ν
TTD6	TTD_Isotropy	-
TTD7	TTD_Homogeneity	-
TTD8	TTD_Elasticity	-
TTD9	TTD_Young	E
TTD10	TTD_Constitutive	$\mathbf{D}$
TTD11	TTD_BodyForce	$\mathbf{b}$

Table 7.5: Theoretical Terminology Definitions for FFEMP

7.1.2.4 Common Theoretical Assumptions

This section presents common theoretical assumptions. Sometimes, it is difficult to distinguish between terminology definitions and assumptions. The following rule is proposed to distinguish between them. Assumptions, including common assumptions and variable assumptions, which are discussed in the *Variable Assumptions* section, should not use any terminology that is not defined in the *Theoretical Terminology Definitions* section. By using

this rule, common theoretical assumptions become simpler. Only a few sentences will be enough to specify them. For example, a common theoretical assumption for FFEMP is that all materials in the computational domain behaves entirely linear elastically (CTA_Elastic), where the terminology elasticity (TTD_Elasticity) is defined somewhere else. This rule is also applied to computational terminology definitions and common computational assumptions when the computational model is documented (Section 7.1.3).

7.1.2.5 Theoretical Model

This section specifies the theoretical model. The mathematical symbols used in the equations, and the terminology the symbols represent, should be defined in the *Theoretical Terminology Definitions* section. The input and the output of the model are also given to improve understandability.

The essential part of a theoretical model is the mathematical equations. The equations of the theoretical model for FFEMP were given in Section 2.4 as Equation 2.1 and discussed in Section 4.3. The theoretical model for FFEMP is shown in Figure 7.6.

7.1.2.6 Variable Assumption

This section lists all variable assumptions. A variable assumption is specified in terms of a feature and its possible values. The possible values are specified by a set, which can be expressed by enumerating all its elements for a small set or by giving its properties for a larger set.

Variable assumptions are documented in tables. The fields in the ta-

TM: When the common theoretical assumptions hold, the theoretical model for FFEMP that refines the functional goals is a set of Partial Differential Equations as shown in Equation 2.1.
The input of the model is

- the computational domain (Ω)
- the body force ($\mathbf{b}$)
- the material properties (E and ν)
- the applied load $\mathbf{t}^{(\hat{\mathbf{n}})}$ on surface S_T
- displacement constraints $\mathbf{u} = \bar{\mathbf{u}}$ on surface S_U

The output of the model is

- the displacement $\mathbf{u}$
- the strain $\boldsymbol{\epsilon}$
- the stress $\boldsymbol{\sigma}$

Figure 7.6: The Theoretical Model for FFEMP

ble include Number, Label, Related Items, Description, Variants and History, which are discussed in Section 7.1.1 and Type, which is discussed in Section 7.1.2.3.

An example of the table format for a variable assumption for FFEMP is shown in Table 7.6. A detailed description is lengthy and is not given here.

Number	**VTA3**
Label	**VTA_StressState**
Type	sequence [6] of $\mathbb{B}$
Related Items	TTD_Stress, VTA_StrainState
Variants	VTA_StressState is an sequence of six boolean variables. Each variable has two possible values, T or F.
Description:	This variable theoretical assumption determines the state of stress for the body.
History	Created - July, 2009

Table 7.6: An Example Variable Assumption for FFEMP

This assumption specifies that the i-th entry of *VTA_StressState* is T if it is known that the i-th entry of the stress (Equation 4.2) is zero. For example, for 3 dimensional problem, $VTA\_StressState = \langle F, F, F, F, F, F \rangle$. Another example is the state of plane stress, where $VTA\_StressState = \langle F, F, T, F, T, T \rangle$, since it is known that $\sigma_{zz} = \tau_{yz} = \tau_{xz} = 0$. A uniaxial state of stress in the x direction would require $VTA\_StressState = \langle F, T, T, T, T, T \rangle$, since it is known in this case that every entries except σ_{xx} is equal to zero. The constraint between *VTA_StressState* and *VTA_StrainState* and other details on VTA_StressState can be found in the TMS for FFEMP (Yu, 2010b).

All variable assumptions for FFEMP are shown in Table 7.7

Number	Label	Type
VTA1	VTA_SelfWeight	$\mathbb{B}$
VTA2	VTA_MultiMaterial	$\mathbb{B}$
VTA3	VTA_StressState	sequence [6] of $\mathbb{B}$
VTA4	VTA_StrainState	sequence [6] of $\mathbb{B}$

Table 7.7: Variable Assumptions for FFEMP

7.1.2.7 Traceabilty Matrix

This section presents the traceability matrix between the theoretical terminology definitions and common theoretical assumptions. The matrix for FFEMP is shown in Table 7.8. The check mark in an entry, such as (TTD_Displace, CTA_Cartesian) indicates TTD_Displace depends on CTA_Cartesian. That is, if CTA_Cartesian changes, then TTD_Displace will change.

	TTD_Displace	TTD_Stress	TTD_Strain	TTD_Isotropy	TTD_Homogeneity	TTD_Elasticity	CTA_Continuum	CTA_Cartesian	CTA_DistribMoment	CTA_InternalStressStrain	CTA_SmallDeform	CTA_Elastic
TTD_Displace								✓				
TTD_Stress							✓	✓	✓	✓		
TTD_Strain	✓						✓	✓	✓	✓	✓	
TTD_LDOpt	✓		✓									
TTD_Poisson			✓									
TTD_Young		✓	✓			✓						
TTD_Constitutive	✓	✓										✓
CTA_Isotropic				✓								
CTA_Homogeneous					✓							
CTA_Elastic						✓						

Table 7.8: Traceability Matrix between Theoretical Terminology Definitions and Common Theoretical Assumptions for FFEMP

7.1.3 Computational Model Specification

The documentation for the computational model is called the Computational Model Specification (CMS). The template of CMS is shown in Figure 7.7. The details on each section are described below. It is assumed that the theoretical model for the program family is developed and documented in TMS and the variable assumptions that should be determined are determined before developing the CMS.

1 Introduction
2 Theoretical Model
3 Computational Technique
4 Computational Terminology Definitions
5 Common Computational Assumption
6 Computational Model

Figure 7.7: The Template for Computational Model Family Specification

7.1.3.1 Introduction

This section gives the purpose and the organization of the document.

7.1.3.2 Theoretical Model

This section gives a simple description, which includes the equations of the theoretical model that the computational model refines for quick reference. The restatement of theoretical models makes the document relatively independent and easy to use.

7.1.3.3 Computational Technique

This section documents the selection of the appropriate computational technique for solving the theoretical model. As discussed in Section 4.4.2, AHP (Saaty, 1980, 2008) is used to make the decision. This section includes

- A list of techniques available to solve the theoretical model.
- A list of nonfunctional goals related to the selection of the techniques.
- A pairwise comparison matrix for the nonfunctional goals.
- Pairwise comparison matrices for the techniques with respect to all nonfunctional goals.

The lists and matrices of selection for FFEMP have been shown in Section 4.4.2. The resulting computational technique and an introduction to the technique are also included in this section.

7.1.3.4 Computational Terminology Definitions

This section defines computational terminology with respect to common computational assumptions, which are documented in the *Common Computational Assumption* section.

Computational terminology definitions are documented in a table format. The fields in the table include Number, Label, Related Items, Description and History, which are discussed in Section 7.1.1 and Type, Unit and Sources, which are discussed in Section 7.1.2.3. Since computational terminology relates to a computational model, the computational details, which are usually

related to design or implementation in other types of software, may be reflected in the type of some computational terminology. For example, the use of the Finite Element Method in FFEMP implies that vectors and matrices are used as types in the computational terminology, since the terms **F**, **K** and **a** in the computational model (Equation 2.2) are either vectors or matrices.

An example of a table format specification of a computational terminology definition for FFEMP is shown in Table 7.9. This table defines the computational terminology displacement.

Number	**CTD6**
Label	**CTD_Displace**
Symbol	**a**
Type	Vector
Units	L
Related Items	TTD_Displace, CCA_Diff, CTD_Node
Sources	Smith and Griffiths (1998), Hughes (2000)
Description	The displacement defined here is general displacements of nodes.
History	Created - Sept., 2009

Table 7.9: The Computational Terminology Definition for Displacement in FFEMP

Another example computational terminology definition, shape function, is shown in Table 7.10. Assume that each node has 2 degrees of freedom. Let $\mathbf{u}^e = \begin{bmatrix} u \\ v \end{bmatrix}$ represent the displacement of any point in an element and

Number	**CTD9**
Label	**CTD_Shape**
Symbol	**N**
Type	Matrix
Units	-
Related Items	TTD_Displace, CCA_Diff, CTD_Node, CTD_Element
Sources	Smith and Griffiths (1998), Hughes (2000)
Description	The shape function is a function of coordinates of a point inside an element. It is the interpolation of displacement anywhere in the element in terms of displacements of nodes. Details are given below.
History	Created - Sept., 2009

Table 7.10: The Definition for the Computational Terminology Shape Function in FFEMP

$\mathbf{a}^e = \begin{bmatrix} u_1 \\ v_1 \\ \vdots \\ u_n \\ v_n \end{bmatrix}$, where n is the number of nodes per element, represent the displacement of nodes, then the shape function is $\mathbf{N}$, such that $\mathbf{u}^e = \mathbf{N}\mathbf{a}^e$.

7.1.3.5 Common Computational Assumptions

This section presents common computational assumptions for refining the theoretical models. The same rule of distinguishing theoretical terminology and theoretical assumptions are applied for distinguishing computational terminol-

ogy and common computational assumptions. Hence, the common computational assumptions are short and simple. An example common computational assumptions for FFEMP is show in Figure 7.8, where **N** and **a** is introduced in Section 7.1.3.4 and **u** is introduced in Section 7.1.2.3.

CCA1 (CCA_Diff): The displacement anywhere in an element can be interpolated in terms of displacements at the nodes of the element. That is $\mathbf{u} = \mathbf{Na}$, where the shape function **N** is differentiable.

Figure 7.8: An Example Common Computational Assumption for FFEMP

7.1.3.6 Computational Model

This section specifies the computational model, which is usually represented by a set of equations. This model depends on the numerical technique, which is selected and documented in the *Computational Technique* section, for solving equations representing the theoretical model.

The symbols used in the equations, and the terminology the symbols represent, should be defined in the *Computational Terminology Definitions* section or the *Theoretical Terminology Definitions* section in TMS. The input and the output of the model are also given to improve the understandability. The computational model for FFEMP is shown in Figure 7.9.

7.1.3.7 Traceability Matrix

This section presents the traceability matrix between the computational terminology definitions and common computation assumptions. The matrix

CM: When the common computational assumptions hold, the computational model for FFEMP that refines the theoretical model is represented by Equation 2.2
The input of the model is

- the mesh (**M**)
- the body force (**b**)
- the material properties (E and ν)
- the traction boundary
- the prescribed displacement boundary

The output of the model is

- the displacement of nodes **a**

Figure 7.9: The Computational Model for FFEMP

for FFEMP is shown in Table 7.11. The check mark in an entry, such as (CTD_Element, CTD_Node) indicates CTD_Element depends on CTD_Node. That is, if CTD_Node changes, then CTD_Element will change.

	TTD_Displace	TTD_Stress	TTD_Strain	TTD_LDOpt	TTD_Constitutive	CTD_Node	CTD_Constraint	CTD_Element	CTD_BoundElm	CTD_Shape	CTD_Kinematic	CTD_Stiff	CTD_Load
CTD_Element						✓							
CTD_BoundElm						✓	✓						
CTD_Mesh								✓	✓				
CTD_Displace	✓												
CTD_Stress		✓											
CTD_Strain			✓		✓								
CTD_Kinematic										✓			
CTD_Stiff				✓							✓		
CTD_Load					✓					✓			
CCA_Diff	✓								✓			✓	✓
CCA_DOF						✓							

Table 7.11: Traceability Matrix between Computational Terminology Definitions and Common Computational Assumptions for FFEMP

7.2 Domain Design

The documents for domain design discussed in this section are used to specify the reference architecture for a program family in FASCS, which include the Reference Module Guide (RMG), which is discussed in Section 7.2.1 and Reference Module Interface Specification (RMIS), which is discussed in Section 7.2.2.

7.2.1 Reference Module Guide

RMG gives module decomposition of the program family. Modules documented in RMG are abstract. Only secrets and services for each module are documented. The template of RMG is shown in Figure 7.10. The details for

Figure 7.10: The Template for Reference Module Guide

each section are given below.

7.2.1.1 Introduction

This section gives the purpose and the organization of the document.

7.2.1.2 Changes

Both Anticipated Changes (ACs) and Unlikely Changes (UCs) are documented as lists with numbers and names. The prefixes are shown in Figure 7.1.

Examples of both changes for FFEMP are given below. AC1 and AC10 are anticipated changes related to evolving the whole program family. AC1 is usually implemented by the operating system and the programming language for implementing the program family. Other changes are implemented by FFEMP. AC18 is an anticipated change related to different members of the program family. UCs are usually related to goals and assumptions developed in the domain requirement engineering stage, except for ones related to the hardware, such as UC1. The relationship is explicitly given in the parentheses that follow the UC's name so that the change can easily to identified.

AC1 (AC_File): The data structure and algorithms for implementing the interface between files and the operating system

AC10 (AC_Node): The data structure of a node

AC18 (AC_NumNode): The number of nodes for an element

UC1 (UC_Input): The input devices are File and Keyboard.

UC3 (UC_Displacement): Given the computational domain, the material

properties and the boundary conditions, FFEMP can solve for the displacement of any point in the domain. (FG_Displacement)

UC4 (UC_Newton): The physics for the problems are in the field of Newton's classical mechanics. (CTA_Newton)

UC16 (UC_Diff): The displacement anywhere in an element can be interpolated in terms of displacements at the nodes of the element. That is $\mathbf{u} = \mathbf{Na}$, where the shape function $\mathbf{N}$ is differentiable. (CCA_Diff)

Anticipated and unlikely changes can be reused. For example, AC1 and the modules to hide AC1, which is a module that is not implemented by FFEMP, are likely the same for many programs. AC10 and the corresponding module to hide AC10 are likely the same for many FEM programs. UC1 can be reused by many programs. UC3 can be reused by solid mechanics program. UC16 can be reused by FEM programs.

7.2.1.3 Module Hierarchy

This section provides an overview of the module design. Modules are in a hierarchy decomposed by secrets. The modules that are leaves in the hierarchy tree are the modules that will actually be implemented. The module hierarchy for FFEMP has been shown in Figure 3.1.

The module decomposition is performed and documented according to Parnas et al. (1984). The highest level of abstraction of the decomposition includes three modules: Hardware-Hiding Module, Behavior-Hiding Module and Software Decision Module. The leaf modules for Hardware-Hiding Module are

commonly used modules and are already implemented by the Operating System where the members of program family are executed and the programming language used to implement the program family. The remaining leaf modules are implemented by the program family.

7.2.1.4 Module Decomposition

Modules are decomposed according to the principle of "information hiding" (Parnas, 1972). A name is given for each leaf module. In addition, a label is assigned for each module for a short identifying phrase. There are three other fields for each module. The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. Only leaf modules in the hierarchy have to be implemented. If a dash (-) is shown, this means that the module is not a leaf and will not have to be implemented. Whether or not this module is implemented depends on the programming language selected. The "implementation" in this section means the implementation in the application implementation stage.

An example module for FFEMP is shown below. This is a leaf module that hides AC10, as discussed in Section 7.2.1.2. It is implemented by FFEMP.

Node Module (M_Node)

Secrets: The data structure of a node

Services: Defining the data structures and providing the operations for nodes.

Implemented By: FFEMP

7.2.1.5 Use Relation

The use relation for modules in a program family is presented in a graph. The use relation for FFEMP is shown in Figure 7.11. The use relation can improve the reusability and maintainability. When a module, say M_Node, needs to change, one can identifies that the modules M_Elm, M_BElm and M_FEM needs to change since they use the module M_Node. This change refers to the change of the secrets and services (interface) of the module, not to internal implementation decision.

7.2.1.6 Traceability Matrix

The traceability matrix between anticipated changes and modules is given in this section. The traceability matrix for FFEMP is partially shown in Table 7.12. This table clearly shows that FFEMP is well designed since all modules have a one to one association with ACs except M_Const and M_FEM. M_Const provides constants associated with variable requirements. Although the calculation of the constants is complicated, the results are simple. Hence, it is maintainable to put these constant together. M_FEM not only provides algorithms associating with the global data structure mesh, it provides operations associating with both the global data structure and local data structure,

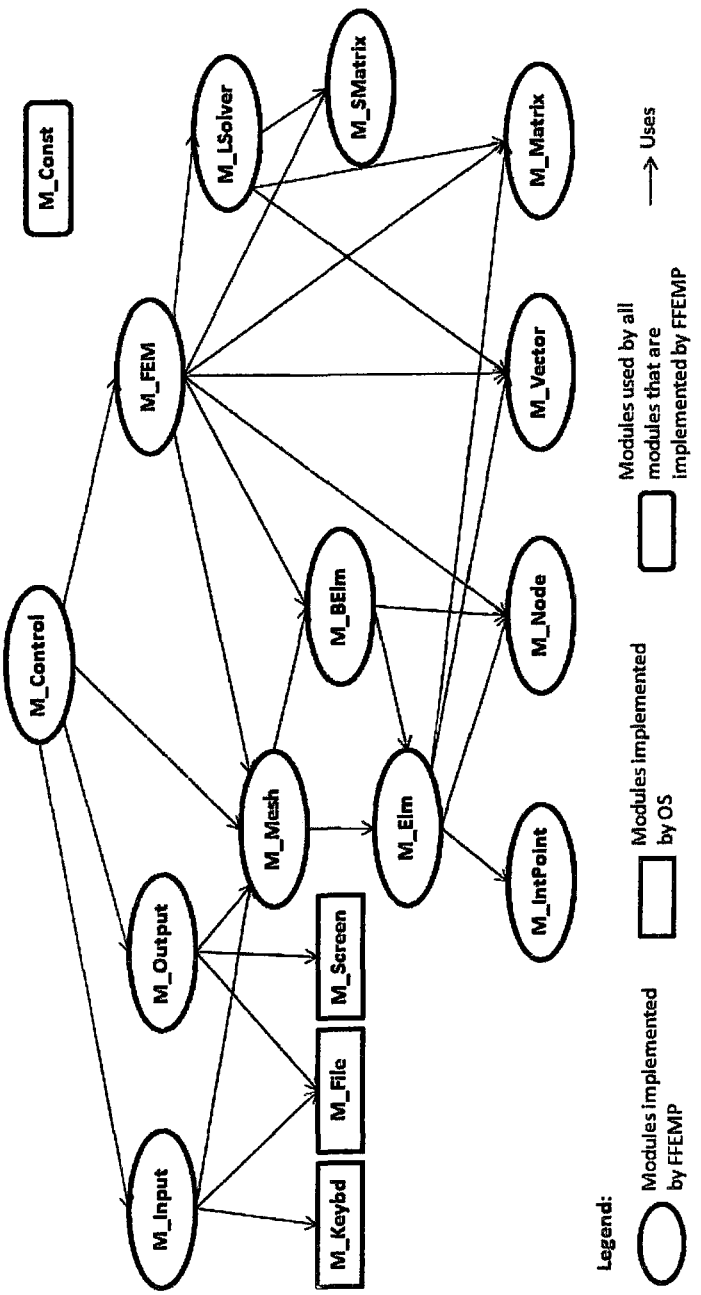

Figure 7.11: Use Relation

	M_Const	M_Vector	M_Matrix	M_SMatrix	M_Node	M_IntPoint	M_Elm	M_BElm	M_Mesh	M_FEM	M_LSolver
AC_Vector		✓									
AC_Matrix			✓								
AC_SMatrix				✓							
AC_Node					✓						
AC_IntPoint						✓					
AC_Elm							✓				
AC_BElm								✓			
AC_Mesh									✓		
AC_LSolver											✓
AC_ElmShape	✓										
AC_NumNode	✓										
AC_IntMethod	✓										
AC_NumIpts	✓										
AC_StressS	✓										
AC_StrainS	✓										
AC_NumBNode	✓										
AC_BIntMethod	✓										
AC_NumBIpts	✓										
AC_Stress	✓										
AC_Strain	✓										

Table 7.12: Traceability Matrix Between Changes and Modules

which includes elements and nodes, as well. The complication of relations with other modules deserves a separated module, even though there is no associated ACs. In addition, two anticipated changes, AC_Stress and AC_Strain, do not associate with any module since the changes do not impact the module decomposition. However, they will affect the interfaces of the module M_Elm, as shown in the Reference Module Interface Specification for FFEMP (Yu, 2010e).

7.2.2 Reference Module Interface Specification

The Reference Module Interface Specification (RMIS) gives the detailed design of the program family. The interface of a module includes the syntax and semantics of access programs in the module. The template for RMIS is shown in Figure 7.12. This temple is adopted from Yu (2007). One difference is the use of environment variables, which are used to specify the interaction between the developed system and the environment, such as input and output devices. Another difference is the variabilities, which are not applicable for a single program, that may impact the interface of the module.

1 Introduction
2 Module Hierarchy
3 Module Interfaces
  3.1 Uses
    3.1.1 Imported Constants
    3.1.2 Imported Data Types
    3.1.3 Imported Access Programs
  3.2 Interface Syntax
    3.2.1 Exported Constants
    3.2.2 Exported Data Types
    3.2.3 Exported Access Programs
  3.3 Interface Semantics
    3.3.1 Environment Variables
    3.3.2 State Variables
    3.3.3 Assumptions
    3.3.4 Invariants
    3.3.5 Access Program Semantics
    3.3.6 Local Constants
    3.3.7 Local Data Types
    3.3.8 Local Functions
    3.3.9 Considerations
  3.4 Variabilities

Figure 7.12: The Template for Documenting RMIS

7.2.2.1 Introduction

This section gives the purpose and the organization of the document.

7.2.2.2 Module Hierarchy

Since RMIS is the detailed specification for the design documented in RMG, the module hierarchy, which is the key of the design, is redisplayed in this section.

7.2.2.3 Module Interfaces

Interfaces for all leaf modules are documented in this section. There are several subsections. When the interface of the module cannot be determined due to the nontermination of the variabilities, information on how to determine the interfaces of a specific family member should be given in the corresponding subsections. An example of how variabilities can impact the interfaces has been given in Section 3.1.2.1.

Uses This sections lists constants, data types and access programs that are used to specify the interface of this module, but are defined outside of this module. The format of each imported item is specified as
Uses ⟨ *module name* ⟩ **Imports** ⟨ *resource* ⟩
where the resource includes constants, data types and access programs.

Interface Syntax This section defines the syntax of the module interface. The interface indicates the services that the module provides. Other modules

can only access this module through this interface. The other information inside the module is the secret that it hides from other modules. Changing this internal information will not affect the way that other modules use this module. The format of documenting the interface syntax and the interface semantic, which will be presented next, is inspired by Hoffman et al. (1995), Yu (2007) and Smith and Yu (2009). The *Interface Syntax* section includes the exported constants, exported data types, and exported access programs. Each access program has a name, input list, output list, and exceptions.

The exported access programs are listed in a table. Some exported access programs in element module (M_Elm) for FFEMP is shown in Table 7.13.

Routine Name	Input	Output	Exceptions
getNode	$\mathbb{N}$	`NodeT`	OutOfBound
setNode	$\mathbb{N}$, `NodeT`		OutOfBound IdenticalNodes
calConst		MatrixT	
calStiff	CoordT	`VectorT`	
calKine	CoordT	`MatrixT`	
calStiff		`MatrixT`	
calStress	`CoordT`, `VectorT`	`VectorT`	

Table 7.13: Some Exported Access Programs for M_Elm

Interface Semantics This section gives the semantics associated with the syntax specified in the *Interface Syntax* section. There are several subsections. The *Environment Variables* give the environment variables that this module may change. Since the modules can be treated as a finite state machine (Hoffman et al., 1995), the state variables for the machine should be given in the

State Variables section. The *Assumption* section presents any assumptions that keep the state machine working properly. The *Invariants* section lists all predicates that should be held for the module.

The *Access Program Semantics* gives the semantics for all access routines. The semantics should be specified as formally as possible to avoid any ambiguity. The semantics of each access routines is specified by the followings:

- **Description**

  Describes the access routine in natural language.

- **Exception**

  Gives exceptions raised by the access routine if applicable.

- **Output** or **Transition**

  It is not recommended that an access routine returning an output and changing the state or environment variables at the same time. Hence, this two items do not appear in the semantics for the same module. Output gives the return value of the access routine, which is a function. Transition presents how state variables or environment variables change.

- **Related Variabilities**

  Gives the variabilities that impact the semantics of the access program and the instruction on how to simplify the semantics, or how to make the the semantics concrete, according to the determined values of the variabilities.

The *Local Constants*, *Local Data Types* and *Local Functions* sections document any constants, data type and functions that are used to express

semantics of the access routines to simplify the presentation of the semantics. The *Considerations* section documents any other issues related to this module, but not applicable to other sections.

The interface semantics for access program calKine in the module M_Elm for FFEMP is partially displayed in Figure 7.13, where *shapemat* and *ldopt* are two local functions. The function *shapemat* (`CoordT` $\longrightarrow$ `MatrixT`) re-

(*outkine*: `MatrixT`)calKine(*inlcoord*: `CoordT`)

- **Description**
  Return the kinematics matrix, which is a derivative of the matrix format of the shape function.
- **Output**
  $outkine := (ldopt(shapemat))(inlcoord)$

Figure 7.13: The Semantics of calKine in M_Elm Module

turns a matrix format of the evaluated shape function and *ldopt* ((`CoordT` $\longrightarrow$ `MatrixT`) $\longrightarrow$ (`CoordT` $\longrightarrow$ `MatrixT`)) is a linear differentiation operator that takes a function returning a matrix as input and returns a function returning a matrix. The semantics of the function *ldopt* is not determined due to the variabilities FVR_StressS and FVR_StrainS as Table 7.14 shows in the next section. The detailed semantics for both local functions can be found in the RMIS for FFEMP (Yu, 2010e).

Another example interface semantics, which is for the access routine calConst, is below.

(*outconst*: `MatrixT`)**calConst()**

- **Description**

  Return the constitutive matrix, which gives the relationship between stress $\boldsymbol{\sigma}$ and the strain $\boldsymbol{\epsilon}$. If the material is linearly elastic, then the constitutive matrix is a constant.

- **Output**

  outconst, such that $\boldsymbol{\sigma} = outconst.\text{mulv}(\boldsymbol{\epsilon})$

- **Related Variabilities**

  The semantics of this function depends on FVR_StressS and FVR_StrainS.

  For a general 3D case: $(E = mat.e \wedge \nu = mat.nu) \wedge$

  $((\text{FRV\_StressS} = \langle F, F, F, F, F, F\rangle \wedge \text{FRV\_StrainS} = \langle F, F, F, F, F, F\rangle)$

  $\Longrightarrow outconst := \mathbf{D}$

  where $\mathbf{D}$ is defined as Equation 3.2.

  The steps to simplify the *outconst* is given below, where $\boldsymbol{\sigma}'$ and $\boldsymbol{\epsilon}'$ represent the stress and strain for a 3D domain.

  1. reduce the size of *outconst* to be the number of Ts in the values of FVR_StressS and FVR_StrainS.
  2. simplify stress $\boldsymbol{\sigma}'$ and strain $\boldsymbol{\epsilon}'$ by following:

     $(\forall i : \mathbb{N})(0 \leq i < 6)(\text{FVR\_StressS}[i] = \text{T} \Longrightarrow \sigma'[i] = 0)$

     $(\forall i : \mathbb{N})(0 \leq i < 6)(\text{FVR\_StrainS}[i] = \text{T} \Longrightarrow \epsilon'[i] = 0)$

  3. obtain stress $\boldsymbol{\sigma}$ and strain $\boldsymbol{\epsilon}$ by removing corresponding entries of $\boldsymbol{\sigma}'[i]$ and $\boldsymbol{\epsilon}'[i]$ if $\boldsymbol{\sigma}'[i] = 0$ or $\boldsymbol{\epsilon}'[i] = 0$

4. obtain the constitutive matrix *outconst*, such that

   $\boldsymbol{\sigma} = outconst.\text{mulv}(\boldsymbol{\epsilon})$

Variabilities The *Variabilities* section lists all variabilities that change the interface of the module. In addition, the access programs that each variability changes are also given. The Variabilities section in M_Elm module is shown in Figure 7.14. There are 7 variabilities impacting 6 exported routines and 1 local function of M_Elm module.

FVR_ElmShape
This variability changes the semantics of the exported routine **getNLCoord**.

FVR_NumNode
This variability changes the semantics of the exported routine **getNLCoord**.

FVR_IntMethod
This variability changes the existence of the exported routine **calIpts**. If FVR_IntMethod = GAUSSQ, then **calIpts** is included in the module. Otherwise, if FVR_IntMethod = DIR, then **calIpts** is not included in the module.

FVR_StressS
This variability changes the detailed expression of the exported routine **calConst**. It also change the semantics of exported routine **calShape** and the local function **ldopt**.

FVR_StrainS
This variability changes the detailed expression of the exported routine **calConst**. It also change the semantics of exported routine **calShape** and the local function **ldopt**.

FVR_Stress
This variability changes the existence of the exported routine **calStress**. If FVR_Stress = T, then **calStress** is included in the module. Otherwise, if FVR_Stress = F, then **calStress** is not included in the module.

FVR_Strain
This variability changes the existence of the exported routine **calStrain**. If FVR_Strain = T, then **calStrain** is included in the module. Otherwise, if FVR_Strain = F, then **calStrain** is not included in the module.

Figure 7.14: The Variabilities for M_Elm Module

7.3 Traceability Matrix

The traceability matrix between common and variable requirements and modules that are implemented by the program family should be given. As an example, the traceability matrix for FFEMP is illustrated in Table 7.14 and 7.15. For displace purpose, the matrix is separated into two tables. In this

	M_Input	M_Outout	M_Control	M_Const	M_Vector	M_Matrix	M_SMatrix	M_Node	M_IntPoint	M_Elm	M_BElm	M_Mesh	M_FEM	M_LSolver
FCR_Displace	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 7.14: Traceability Matrix Between Requirements and Modules (I)

example, Only functional requirements are listed in the first column. Non-functional requirements do not contribute to the design of FFEMP. The first row lists all modules that are implemented in FFEMP.

A check mark in a cell means that the corresponding module of the column should change if the corresponding requirements of the row changes. Table 7.14 shows that if functional common requirement FCR_Displace, which is the requirement to calculate the displacement using $\mathbf{F} = \mathbf{Ka}$, changes, then all modules may change. For example, if the technique for solving the theoretical module were changed from the Finite Element Method (FEM) to using closed formed solutions (DM), modules related to FEM, which include M_Node, M_IntPoint, M_Elm, M_BElm, M_Mesh, M_FEM and M_LSolver would change. This means that FCR_Displace is the basis of other requirements. The change in this context refers to the change of the syntax of the

	M_Input	M_Output	M_Const	M_Elm	M_BElm
FCR_SameElmShape				✓	
FCR_SameNodeNum				✓	
FCR_SameBElmShape					✓
FCR_SameBNodeNum					✓
FCR_FileInput	✓				
FCR_FileOutput		✓			
FVR_Shape			✓		
FVR_NumNode			✓		
FVR_IntMethod			✓	✓	
FVR_NumIpts			✓		
FVR_StressS			✓		
FVR_StrainS			✓		
FVR_NumBNode			✓		
FVR_BIntMethod			✓		✓
FVR_NumBIpts			✓		
FVR_Stress			✓	✓	
FVR_Strain			✓	✓	

Table 7.15: Traceability Matrix Between Requirements and Modules (II)

module.

On the other hand, Table 7.15 shows that except for FCR_Displace, each of functional variable requirements other than FCR_Displace only relates to one module. Table 7.15 also shows that the change of functional variable requirement, which occurs frequently, only relates to three modules, M_Const, M_Elm and M_BElm. In addition, the type of changes in M_Elm and M_BElm is the existence of some access routine. This means that FFEMP is well designed to hide changes, which include changes related to evolving the whole program family and changes related to variabilities.

Chapter 8

Conclusions

Accuracy, precision and efficiency are important for Scientific Computing (SC) software. Other software quality factors, such as reusability, maintainability, usability and reliability, which are discussed in Chapter 1, also contribute to software quality. However, these quality factors are often neglected by developers of SC software, including professional end user developers, who are scientists developing and using SC software to check their theories.

Improved reusability and maintainability can reduce the end users' time to develop software, so that they can concentrate on their research. Usability not only can reduce their time for using a program, but it can also increase the chance for the end users to use the existing programs. Reliability is also important, especially for some theories that are represented by models that cannot be verified without the use of a computer. It is hard to prove the correctness of theories with an unreliable program.

This work is dedicated to improve the above software quality factors for

SC software since there is room for improving these quality factors, as discussed in Chapter 1. How to adapt existing software engineering methodologies to address the characteristics of end user developers and the characteristics of scientific problems is illustrated. A proof of concept program family, FFEMP, which can solve elasticity problems in solid mechanics using the Finite Element Method (FEM), is developed to show how the proposed methodology can be used.

In this final chapter of the thesis, the major contributions are first summarized in Section 8.1. This is followed by suggested future work in Section 8.2.

8.1 Contributions

In this research, a new methodology, Family Approach for developing Scientific Computing Software (FASCS), is proposed. The family approach, which can improve the reusability, maintainability, usability and reliability, is not new for software development. However, it is the first time in the SC software development community, where all stages in both domain engineering phase and application engineering phase are included. The completeness of the proposed methodology is important for end user developers since they often have little software engineering knowledge.

In addition to the breath, the depth of the family approach for developing SC software is explored. Some new aspects are added to FASCS to distinguish it from other existing family approach methodologies as follows:

- FASCS uses a new methodology, Goal Oriented Commonality Analysis to elicit, analysis and document common and variable requirements.

- An environment for developing members of a program family, named the Family Member Development Environment (FMDE), is systematically divided into sub-environments and proper names are assigned. An FMDE includes a Domain Model (DM), a Variable Code Generator (VCG), a Test Case Generator (TCG), a Family Member Assembler (FMA) and a Family Member Generation Process (FMGP).

- A new test technique, named Computational Variability Test (CVT), is proposed. CVT is a technique to test the common portion of routines that may vary for some members of the program family. It can also be used for the integration test of a program family in the domain engineering phase. This test technique addresses the unknown solution challenge. That is, knowledge of the true solutions is not required for using CVT.

- Documentation is emphasized by FASCS. New templates of documents for the domain requirement engineering stage, including Common and Variable Requirement Specification, Theoretical Model Specification and Computational Model Specification, and for the domain design stage, including Reference Module Guide and Reference Module Interface Specification, are proposed.

The details of the above major contributions on the improvement of software quality factors are summarized in the rest of this section. Other minor ones are mentioned in previous chapters when the FASCS is specified

and are not summarized in this chapter. The challenges mentioned in Section 2.2.1 are also addressed.

8.1.1 Goal Oriented Commonality Analysis

Goal Oriented Commonality Analysis (GOCA) improves reusability, maintainability and reliability. It also addresses the technique selection challenge and modification challenge.

- GOCA proposes two layers of modeling, including the theoretical model and the computational model, to resolve the conflict between the continuous mathematical models that represent the underlying theories of SC problems and the discrete nature of a computer. This conflict, which occurs often in SC software, has apparently not been emphasized in other software development methodologies. The separation of theoretical model and computational model also improves reusability since the theoretical model can be completely reused by programs that solve the same problems, but use different numerical techniques.

- Nonfunctional related artifacts, such as nonfunctional goal and nonfunctional requirements, are difficult to quantify. A decision making technique, the Analytic Heuristic Process (AHP) (Saaty, 1980, 2008), is adapted to rank nonfunctional goals and select the appropriate technique to solve the theoretical model. AHP can also be used to rank nonfunctional requirements to select computation related algorithms or packages. This address the technique selection challenge.

- Assumptions applied to the refinement from goals to a theoretical model and the refinement from a theoretical model to a computational model are explicitly given by using GOCA. This can improve reusability and maintainability, since changes of models can be traced to the changes of the assumptions. These assumptions serve as the key to systematically managing the changes of models and requirements. Since the assumptions are the major source of the modification for programs used to check theories under development, explicitly specifying the assumptions addresses the modification challenge.

- Models are usually represented by mathematical equations. Terminology to specify the models, as well as common and variable requirements, are defined and documented formally to avoid ambiguity that may reduce the reliability.

- The different levels of abstractions for terminology, which is based on the classification of common and variable assumptions, defines different levels of the theoretical model and computational model. The common assumptions improve reusability, since the terminology and models that are defined under common assumptions are abstract and can be completely reused by a relatively large set of programs. The variable assumptions also improve reusability, since by choosing different values of a variable assumption, not only the terminology definitions and models can be completely reused, the whole program family, including the requirements, the design, the code and the test cases, can be reused with some mi-

nor changes. For example,the variable assumption VTA_MultiMaterial (VTA4), which is listed in Table 7.7, gives developers freedom to choose whether the program family can handle the computational domain with multiple materials. If VTA4 changes, only the element module M_Elm, the constant module M_Const and the input module M_Input need to change. Other part of the program family stays the same.

8.1.2 Family Member Development Environment

This thesis gives explicit instructions on how to develop a Family Member Development Environment (FMDE) for SC program families. The use of FMDE improves reusability and reliability and addresses the modification challenge, input output challenge and unknown solution challenge.

- FMDE can be used to automatically generate a specific member of a program family to improve reusability. The automation of the generation addresses the characteristic of end user developers of SC software who usually do not have much software engineering knowledge. It also addresses the characteristic of SC problems that the generation of variable code requires a large amount of computation. In addition, the automation allows end user developers to quickly generate specific family members, which addresses the modification challenge.

- A DM defines a Domain Specific Language (DSL) for the program family. Writing a program in the DSL, which is the input to VCG, TCG and FMA is much easier than writing the variable part of the code and the

test cases. Hence, the FMDE improves reusability.

- Testing SC program families are rarely mentioned in developing SC program family literature. Test cases for SC program families have apparently never been automatically generated. Systematic testing can improve reliability. The new TCG can automatically generate the variable part of some benchmark test cases. The automation addresses the input output challenge since complicated input data can be automatically generated. The automation also make the use of the Computational Variability Test (CVT) to test a program family feasible. CVT addresses the unknown solution challenge since no true solutions are required for using CVT.

8.1.3 Documentation

Documentation is emphasized in FASCS, which can improve reusability, maintainability and usability. The input output challenge and the modification challenge are also addressed. Although documentation does not directly address the approximation challenge, it provide facilities, such as formal definitions of terminology and explicit assumptions for models, to perform error analysis.

- The theoretical model and computational model are documented separately to improve reusability.

- In addition to the assumptions, which may change and impact models and requirements as discussed above, changes that effect the modular

design of the program family are explicitly documented. The modular design can improve reusability and maintainability, since understandability is improved. The ability to trace current and potential changes can also improve reusability and maintainability.

- A graphic notation for functional variable requirements of the program family is included. This notation improves reusability and maintainability since it improves the understandability. The idea of the graphic notation is not new for developing program families. However, a new notation that addresses the input output challenge is added to the graph. The new notation makes the representation of excessive input data concise.

- The instruction on how to develop module interfaces that vary for some family members is explicitly given in the RMIS. This can help application engineers with quickly developing module interfaces for a specific family member, which in turn improves reusability. The modification challenge is thus addressed.

- Traceability matrices can improve reusability and maintainability. The traceability matrices in FFEMP include:

  - traceability matrices for the domain requirement engineering stage, which include goals, terminology definitions, assumptions, models, and requirements

  - traceability matrices for the domain design stage, which include changes, variable requirements and modules.

- traceability matrices between requirements, modules, code and test cases

8.2 Future Works

The results of current work encourages further research in the field of using a family approach to develop SC software. The suggested investigations needed to evaluate the effectiveness of our work are as follows:

- Improve FFEMP.
  - Although the requirements and design for the current version of FFEMP include 3D, the implementation and testing is only for 2D. The 3D implementation and testing can be conducted to make FFEMP more practical.
  - More variabilities can be added to facilitate solving a larger set of problems, such as solving plasticity problems.
- Develop more program families in other domains, such as for financial applications, which may have different theoretical models and computational models, as case studies to provide more feedbacks for the further improvement of FASCS.
- Establish a methodology for testing nonfunctional requirements and perform tests on reusability, maintainability, usability and reliability of example program families developed using FASCS. Compare these tests

with tests for similar single programs and obtain the quantitative data on the improvement of using FASCS.

- Extend GOCA so that the computational model does not depend on the theoretical model. Hence, the computational model is more abstract and more reusable. For example, instead of solving the theoretical model using Finite Element Method (FEM), the computational model can be design to solving general Partial Differential Equations using FEM. In this case, the computational model is more reusable.

- Increase the use of tools to support the automation. For example, tools can be adopted to automatically generate documents for a specific family member.

- Explore the use of Computational Variability Test (CVT). Changing computational variabilities can be used in a manner analogous to grid refinement studies. In grid refinement a series of increasing dense grids is compared to determine whether the solution is converging. In some cases a theoretical convergence rate is available that can be compared with the actual convergence rate as an additional check, as described in Roache (1998). Increasing the the order of interpolation of the finite elements can be used in the same way.

Bibliography

Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

Yuri Alexeev, Benjamin A. Allan, Robert C. Armstrong, David E. Bernholdt, Tamara L. Dahlgren, Dennis Gannon, Curtis L. Janssen, Joseph P. Kenny, Manojkumar Krishnan, James A. Kohl, Gary Kumfert, Lois Curfman Mcinnes, Jarek Nieplocha, Steven G. Parker, Craig Rasmussen, and Theresa L. Windus. Component-based software for high-performance scientific computing. *Journal of Physics: Conference Series*, 16:536–540, 2005.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. Lapack users' guide third edition, 22 Aug 1999.

Ritu Arora, Purushotham Bangalore, and Marjan Mernik. Developing scientific applications using generative programming. In *SECSE '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 51–58, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3737-5. doi: http://dx.doi.org.libaccess.lib.mcmaster.ca/10.1109/SECSE.2009.5069162.

Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume ii: Technical concepts of component-based software engineering, 2nd edition. Technical Report CMU/SEI-2000-TR-008 ESC-TR-2000-007, Software Engineering Institute, Carnegie Mellon University, May 2000.

María Cecilia Bastarrica and Nancy Hitschfeld-Kahler. Designing a product family of meshing tools. *Advances in Engineer Software*, 37(1):1–10, 2006. ISSN 0965-9978. doi: http://dx.doi.org/10.1016/j.advengsoft.2005.04.001.

Nelson R. Bauld. *Mechanics of Materials.* PWS Publishers, 1986.

Kent Beck, Retrieved 2010. URL `http://www.junit.org/`. JUnit.org Resources for Test Driven Development.

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

Guntram Berti. Gral-the grid algorithms library. *Future Generation Computer Systems*, 22(1-2):110–122, 2006. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2003.09.002.

BLAS. Blas (basic linear algebra subprograms), Retrieved December 2010.

Charles Blilie. Patterns in scientific software: An introduction. *Computing in Science and Engineering*, 4(3):48–53, 2002. ISSN 1521-9615. doi: http://dx.doi.org/10.1109/5992.998640.

Blitz. Blitz++, object-oriented scientific computing, Retrieved December 2001. URL `http://www.oonumerics.org/blitz/`.

Kevin Burr and William Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of the International Conference on Software Testing Analysis and Review*, pages 503–513. West, 1998.

Jacques Carette. Gaussian elimination: a case study in efficient genericity with metaocaml. *Science of Computer Programming*, 62(1):3–24, 2006. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2005.10.012.

Jacques Carette, Mustafa Elsheikh, and Spencer Smith. A generative geometric kernel. In *ACM SIGPLAN 2011 Workshop on Partial Evaluation and Program Manipulation (PEPM'11)*, 2011.

Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software development environments for scientific and engineering software: A series of case studies. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 550–559, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.77.

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

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

Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional., Boston, MA, USA, 2002.

CppUnit, Retrieved 2010. URL `http://sourceforge.net/projects/cppunit/`. CppUnit - C++ port of JUnit.

Carlton A. Crabtree, A. Gunes Koru, Carolyn Seaman, and Hakan Erdogmus. An empirical characterization of scientific software development projects according to the boehm and turner model: A progress report. In *SECSE '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 22–27, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3737-5. doi: http://dx.doi.org/10.1109/SECSE.2009.5069158.

K. Czarnecki and U.W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

Timothy Davis, Last Access 2010. CSPARSE - A Concise Sparse Matrix Package in C.

Viktor K. Decyk and Henry J. Gardner. Object-oriented design patterns in fortran 90/95: mazev1, mazev2 and mazev3. *Computer Physics Communications*, 178(8):611 – 620, 2008. ISSN 0010-4655. doi: DOI: 10.1016/j.cpc.2007.11.013.

P. Di Felice. Reusability of mathematical software: A contribution. *IEEE Transactions on Software Engineering*, 19(8):835–843, 1993. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.238586.

Paul F. Dubois. Designing scientific components. *Computing in Science and Engineering*, 4(5):84–90, 2002. ISSN 1521-9615. doi: http://dx.doi.org/10.1109/MCISE.2002.1032434.

Steve M. Easterbrook and Timothy C. Johns. Engineering the software for understanding climate change. *IEEE Des. Test*, 11(6):65–74, 2009. ISSN 0740-7475. doi: http://dx.doi.org/10.1109/MCSE.2009.193.

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

Mustafa Elsheikh. A generative approach to meshing geometry. Master's thesis, McMaster University, 2010.

Efstratios Gallopoulos, Elias Houstis, and John R. Rice. Computer as thinker/doer: Problem-solving environments for computational science. *Computing in Science and Engineering*, 1:11–23, 1994. ISSN 1070-9924. doi: http://doi.ieeecomputersociety.org/10.1109/99.326669.

E. Gamma, R. Helm, J. Vlissides, and I R Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

GSL. Gsl - gnu scientific library, Retrieved December 2010.

Bruno Harbulot and John R. Gurd. Using aspectj to separate concerns in parallel scientific java code. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 122–131, New York, NY, USA, 2004. ACM. ISBN 1-58113-842-3. doi: http://doi.acm.org/10.1145/976270.976286. URL `http://doi.acm.org/10.1145/976270.976286`.

Michael T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill Higher Education, 2003. ISBN 0070276846.

George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70485-4.

Daniel Hoffman, , Daniel Hoffman, Paul Strooper, and I Background. *Software Design, Automated Testing, and Maintenance A Practical Approach*. Intl Thomson Computer pr (Sd), 1995. ISBN 978-1850322061.

Elias Houstis, Efstratios Gallopoulos, Randall Bramley, and John Rice. Problem-solving environments for computational science. *IEEE Computer in Science and Engineering*, 4(3):18–21, 1997. ISSN 1070-9924. doi: http://dx.doi.org/10.1109/MCSE.1997.615427.

Tomas J. R. Hughes. *The finite element method: linear static and dynamic finite element analysis*. Dover Publications, Inc., 31 East 2nd Street, Mineola, N.Y. 11501, USA, 2000. ISBN 0486411818.

The Trustees of Indiana University. Generic programming, 2010. URL `http://www.generic-programming.org/`.

Init. My choice, my decision, Retrieved December 2010. URL `http://www.123ahp.com/`.

John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, ISCOPE '97, pages 249–256, London, UK, 1997. Springer-Verlag. ISBN 3-540-63827-X. URL `http://portal.acm.org/citation.cfm?id=646893.709568`.

David Kane, Moses Hohman, Ethan Cerami, Michael McCormick, Karl Kuhlmman, and Jeff Byrd. Agile methods in biomedical software development: a multi-site experience report. *BMC Bioinformatics*, 7(1):273, 2006. doi: 10.1186/1471-2105-7-273. URL `http://www.biomedcentral.com/1471-2105/7/273`.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

Diane Kelly, Nancy Cote, and Terry Shepard. Software engineers and nuclear engineers: Teaming up to do testing, 2010.

Diane F. Kelly. A software chasm: Software engineering and scientific computing. *IEEE Software*, 24:120, 118–119, 2007. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2007.155.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming, ECOOP'97*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

Alexander Kott, William Boag, and Luis Vargas. Analytical hierarchy process in requirements analysis, 1996. URL `http://www.interlog.com/~vacuvox/`.

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

LaTeX. Latex – a document preparation system, Retrieved July 2010. URL `http://www.latex-project.org/`.

Lie-Quan Lee and Andrew Lumsdaine. Generic programming for high performance scientific applications. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 112–121, New York, NY, USA, 2002. ACM. ISBN 1-58113-599-8. doi: http://doi.acm.org/10.1145/583810.583823.

B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21:466–471, June 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359511.359522. URL `http://doi.acm.org/10.1145/359511.359522`.

Maplesoft, Retrieved 2010. Maplesoft, division of Waterloo Maple, Inc. home website, http://www.maplesoft.com/.

Luiz Fernando Martha. An object-oriented framework for finite element programming. In *WCCM V Fifth World Congress on Computational Mechanics*, 2002.

Mathwork, Last Access 2010. MATLAB - The Language Of Technical Computing.

James McCaffrey. Test run: The analytic hierarchy process. WWW page, 2005. URL `http://msdn.microsoft.com/en-us/magazine/cc163785.aspx`. (MSDN magazine).

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

John McCuchan. A generative approach to a virtual material testing laboratory. Master's thesis, McMaster University, September 2007.

John D. McGregor. Testing a software product line. Technical report, Carnegie Mellon Software Engineering Institute, December 2001. CMU/SEI-2001-TR-022, ESC-TR-2001-022.

Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0136290493.

Greg Miller. SCIENTIFIC PUBLISHING: A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science*, 314(5807):1856–1857, 2006. doi: 10.1126/science.314.5807.1856. URL `http://www.sciencemag.org`.

NAG. Numerical algorithms group, Retrieved December 2010.

Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3): 389–415, 2007. ISSN 1066-8888. doi: http://dx.doi.org/10.1007/s00778-007-0044-3.

D L Parnas and P C Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986. ISSN 0098-5589.

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

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

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

B. Patzák, 2000. OOFEM project home page, http://www.oofem.org.

B. Patzák and Z. Bittnar. Design of object oriented finite element code. *Adv. Eng. Softw.*, 32(10-11):759–767, 2001a. ISSN 0965-9978. doi: http://dx.doi.org/10.1016/S0965-9978(01)00027-8.

B. Patzák and Z. Bittnar. Design of object oriented finite element code. *Advances in Engineering Software*, 32(10-11):759–767, 2001b. ISSN 0965-9978. doi: http://dx.doi.org/10.1016/S0965-9978(01)00027-8.

Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automatic and scalable t-wise test case generation strategies for software product lines. In *International Conference on Software Testing (ICST)*, Paris, France, April 2010. IEEE.

P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000. ISBN 0134376331.

Klaus Pohl, Gunter Bockle, and Frank van der Linden. *Software Product Line Engineering*. Springer, 2005.

John R. Rice and Ronald F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Computing in Science and Engineering*, 3(3):44–53, 1996. ISSN 1070-9924. doi: http://dx.doi.org/10.1109/99.537091.

P J Roache. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, New mexico, 1998.

Thomas L. Saaty. Decision making with the analytic hierarchy process. *International Journal of Services Sciences*, 1(1):83–98, 2008.

T.L. Saaty. *The Analytic Hierarchy Process, Planning, Piority Setting, Resource Allocation*. McGraw-Hill, New york, 1980.

Rebecca Sanders and Diane Kelly. Dealing with risk in scientific software development. *IEEE Softw.*, 25:21–28, July 2008. ISSN 0740-7459. doi: 10.1109/MS.2008.84. URL `http://portal.acm.org/citation.cfm?id=1383046.1383169`.

Judith Segal. Models of scientific software development. In *Proceeding 2008 Workshop Software Engineering in Computational Science and Engineering*, 2008.

Judith Segal and Chris Morris. Developing scientific software. *IEEE Softw.*, 25(4):18–20, 2008. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2008.85. URL `http://cs.ua.edu/~SECSE08/Papers/Segal.pdf`.

Carnegie Mellon SEI. A framework for software product line practice, version 5.0. WWW page, Retrieved June 2010. URL `http://www.sei.cmu.edu/productlines/frame_report/index.html`.

I. M. Smith and D. V. Griffiths. *Programming the Finite Element Method*. John Wiley & Sons, Inc., New York, NY, USA, 1998. ISBN 0471965421.

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

S. Smith and W. Yu. A document driven methodology for developing a high quality parallel mesh generation toolbox. *Advances in Engineering Software*, 40(11):1155–1167, 2009. ISSN 0965-9978. doi: http://dx.doi.org/10.1016/j.advengsoft.2009.05.003.

Spencer Smith. Systematic development of requirements documentation for general purpose scientific computing software. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 205–215, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2555-5. doi: http://dx.doi.org/10.1109/RE.2006.61.

Spencer Smith, John McCutchan, and Jacques Carette. Commonality analysis for a family of material models. Unpublished, May 2009.

Spencer Smith, John Mccutchan, and Fang Cao. Program families in scientific computing. Unpublished, 2010.

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

R. V. Southwell. *An Introduction to the Theory of Elasticity for Engineers and Physicists.* Oxford University Press, 1941.

Dieter F. E. Stolle. Lecture notes for ce703: An introduction to the finite element method, mcmaster university, 2008.

Jin Tang. Developing scientific computing software: Current process and future process. Master's thesis, McMaster University, 2008.

Rachid Touzani. An object-oriented framework for finite element programming. In *WCCM V Fifth World Congress on Computational Mechanics*, 2002.

Engin Uzuncaova, Sarfraz Khurshid, and Don Batory. Incremental test generation for software product lines. *IEEE Transactions on Software Engineering*, 36:309–322, 2010. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2010.30.

Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer, Berlin, 2007. ISBN 978-3-540-71436-1.

Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE '01: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, page 249, Washington, DC, USA, 2001. IEEE Computer Society.

Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. *Software Product Line Conference, International*, 0:233–242, 2007. doi: http://doi.ieeecomputersociety.org/10.1109/SPLINE.2007.23.

David M. Weiss and Chi Tau Robert Lai. *Software Product Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

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

William A. Wood and William L. Kleb. Exploring xp for scientific research. *IEEE Softw.*, 20(3):30–36, 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1196317.

Roger Young and Ian MacPhedran. Internet finite element resources, Retrieved January 2011. URL `http://homepage.usask.ca/~ijm451/finite/fe_resources/`.

Wen Yu. A document driven methodology for improving the quality of a parallel mesh generation toolbox. Master's thesis, McMaster University, 2007.

Wen Yu. Common and variable requirement specification for ffem, December 2010a. URL `http://www.cas.mcmaster.ca/~smiths/WenYuThesisFiles/App1CVRS.pdf`.

Wen Yu. Theoretical model specification for ffem, December 2010b. URL `http://www.cas.mcmaster.ca/~smiths/WenYuThesisFiles/App2TMS.pdf`.

Wen Yu. Computational model specification for ffem, December 2010c. URL `http://www.cas.mcmaster.ca/~smiths/WenYuThesisFiles/App3CMS.pdf`.

Wen Yu. Reference module guide for ffem, December 2010d. URL `http://www.cas.mcmaster.ca/~smiths/WenYuThesisFiles/App4RMG.pdf`.

Wen Yu. Reference module interface specification for ffem, December 2010e. URL `http://www.cas.mcmaster.ca/~smiths/WenYuThesisFiles/App5RMIS.pdf`.

Wen Yu and Spencer Smith. Reusability of fea software: A program family approach. In *SECSE '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 43–50, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3737-5. doi: http://dx.doi.org/10.1109/SECSE.2009.5069161.

Appendix A

Shape Function Computation

This Appendix presents how a shape function and its derivatives for the program family FFEMP are computed. Some notations are given before the presentation.

Let n represent the number of node per element and m represent the number of DOFs. Let s be a local coordinate and $\mathbf{N} = [\mathrm{N}_1(s)\ \mathrm{N}_2(s)\ \cdots\ \mathrm{N}_n(s)]$ represent the shape function, where each entry in $\mathbf{N}$ is a function of s and s may have more than one component. For example, if the computational domain is 2D, then s is represented by (s_ξ, s_η) and the directions of ξ and η are orthogonal. Let $\mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}$ represent the values of the some type of DOFs, such as the displacement in the x direction, for the nodes of an element. Let d_s represent the type of DOF that is the same as $\mathbf{d}$ for the point with coordinate s.

According to the definition in the Reference Module Interface Specification (RMIS) (Yu, 2010e), the Equation A.1 holds.

$$d_s = \mathbf{N}\mathbf{d} \tag{A.1}$$

In practice, the shape function is often approximate by a function defined in Equation A.2, where s_i represents the local coordinate for node i.

$$N_i(s_j) = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{A.2}$$

Many types of functions, such as Lagrange functions, can be used.

Polynomials are used as an illustration. That is, each entry of the shape function is represent by a polynomial. The degree of polynomial depends on the number of nodes per element and the type of the element. For example, some polynomials for 2D shape functions can be represents by Pascal triangles (Hughes, 2000) as shown in Figure A.1, A.2 and A.3 and the corresponding types of elements are shown in Figure A.4, A.5 and A.6, respectively.

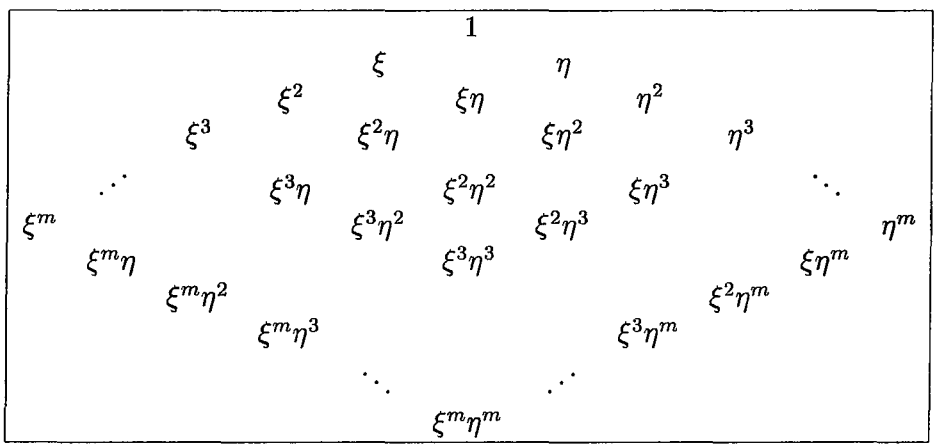

Figure A.1: Pascal Triangle for Lagrange Quadrilaterals (Hughes, 2000)

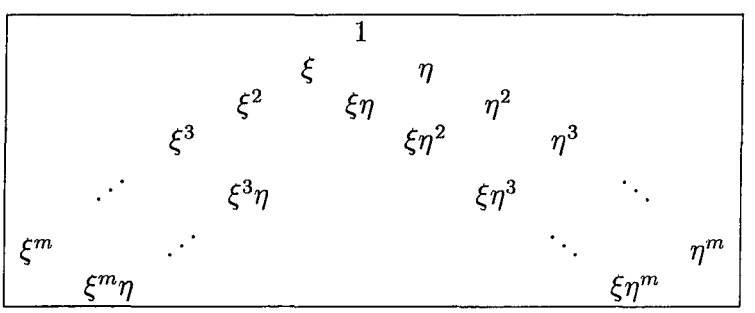

Figure A.2: Pascal Triangle for Serendipity Quadrilaterals (Hughes, 2000)

For illustration purpose, take the computation of 1D shape function as an example. For 1D, the degree of the polynomial is $n-1$. Let c_{ij} represent the the coefficient of s^j for the N_i, then

$$N_i(s) = c_{i0} + c_{i1} * s + \cdots + c_{i(n-1)} * s^{n-1} \tag{A.3}$$

$$
\begin{array}{ccccccccccc}
 & & & & 1 & & & & & & \\
 & & & \xi & & & \eta & & & & \\
 & & \xi^2 & & \xi\eta & & & \eta^2 & & & \\
 & \xi^3 & & \xi^2\eta & & \xi\eta^2 & & & \eta^3 & & \\
\iddots & & \iddots & & \xi^2\eta^2 & & \ddots & & & \ddots & \\
\xi^m & \xi^{m-1}\eta & & \xi^{m-2}\eta^2 & \cdots & \xi^2\eta^{m-2} & & \xi\eta m-1 & & & \eta^m
\end{array}
$$

Figure A.3: Pascal Triangle for Lagrange Triangles (Hughes, 2000)

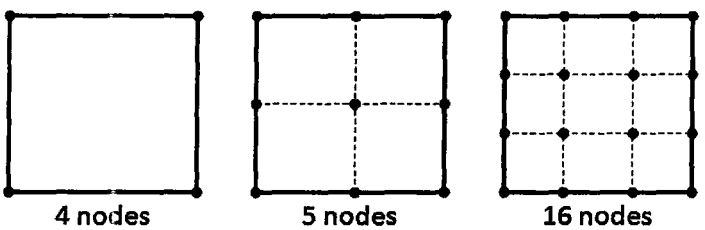

Figure A.4: Lagrange Quadrilateral

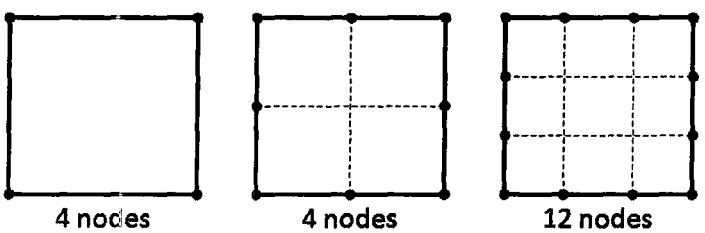

Figure A.5: Serendipity Quadrilateral

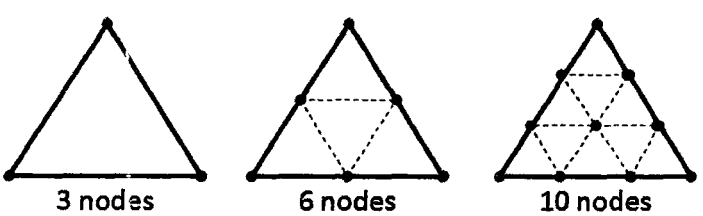

Figure A.6: Lagrange Triangle

Combining Equation A.3 and Equation A.2 we can get a system of n linear equations with n unknown coefficients, as shown in Equation A.4.

$$c_{i0} + s_1 * c_{i1} + \cdots + (s_1)^{n-1} * c_{i(n-1)} = 0 \tag{A.4a}$$

$$\cdots$$

$$c_{i0} + s_i * c_{i1} + \cdots + (s_i)^{n-1} * c_{i(n-1)} = 1 \tag{A.4b}$$

$$\cdots$$

$$c_{i0} + s_n * c_{i1} + \cdots + (s_n)^{n-1} * c_{i(n-1)} = 0 \tag{A.4c}$$

Written in matrix format, Equation A.4 becomes Equation A.5, where

$$\mathbf{S} = \begin{bmatrix} 1 & s_1 & \dots & (s_1)^{n-1} \\ & & \cdots & \\ 1 & s_i & \dots & (s_i)^{n-1} \\ & & \cdots & \\ 1 & s_n & \dots & (s_n)^{n-1} \end{bmatrix} \text{ and } \mathbf{e}_i \text{ is the } i\text{-th column of identity matrix.}$$

$$\mathbf{S}\mathbf{c_i} = \mathbf{e}_i \tag{A.5}$$

The calculated coefficients for $N_i(s)$ are stored in the vector $\mathbf{c}_i = \begin{bmatrix} c_{i0} \\ c_{i1} \\ \vdots \\ c_{i(n-1)} \end{bmatrix}$.

To obtain the shape function $\mathbf{N}$, we need to solve n such systems of equations. That is, we need to solve Equation A.6, where $\mathbf{I}$ is the identity matrix.

$$\mathbf{SC} = \mathbf{I} \tag{A.6}$$

The matrix $\mathbf{C} = [\mathbf{c}_1 \ \cdots \ \mathbf{c}_n]$ stores all coefficients of the shape function $\mathbf{N}$.

Since the Variable Code Generator (VCG) for FFEMP is implemented in Matlab (Mathwork, Last Access 2010), the derivatives can be computed by the symbolic computation tool box in Matlab. It also can be computed numerically by the Chain Rule, which will not be further discussed in this appendix. The VCG for FFEMP computes the derivatives numerically to check the possibility of numerically computing the derivatives of shape functions for simple type of element. It could be changed to using symbolic computation.