DOCUMENTATION DRIVEN TESTING OF SCIENTIFIC COMPUTING SOFTWARE

DOCUMENTATION DRIVEN TESTING OF SCIENTIFIC COMPUTING SOFTWARE

By

BINGZHOU ZHENG, B.ENG.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2009) McMaster University

(Computer Science) Hamilton, Ontario

TITLE: Documentation Driven Testing of Scientific Computing Software

AUTHOR: Bingzhou Zheng, B.Eng. (Taiyuan University of Technology)

SUPERVISOR: Dr. Ned. Nedialkov

NUMBER OF PAGES: vii, 104

Abstract

Domain experts, who create mathematical models and then implement scientific computing software typically focus on their models and implementation, but often pay less attention to systematic and extensive testing of their software. One reason for this situation is that domain experts usually lack software testing experience and know little about testing tools. However, it is desirable to introduce software testing techniques and tools into the development process of scientific computing software.

This thesis originates from testing the scientific computing package DAETS, Differential-Algebraic Equations by Taylor Series. Documentation driven testing, code coverage analysis, and software testing tools are utilized to help verify and improve the quality of the software in this testing project. First, static inspection is used to check the correctness and completeness of the user guide, and verify the consistency of public interface information between the user guide and the source code. Then, black box test cases are designed based on public interface specifications in the user guide. After that, by executing code coverage analysis, test cases are added based on white box testing. Finally, the automatic testing framework tool CppUnit is used to automate the testing process, which greatly facilitates regression testing.

In the DAETS testing projects, 163 test cases (more than 5000 line test code) are implemented, 27 documentation and software defects are found, and 150 lines of dead code are removed.

Acknowledgments

I would like to start by thanking my supervisor, Dr. Ned Nedialkov who gave me the great opportunity to study and do research here. I wish to express my sincere gratitude for his strong support, constant encourage, and instructive guidance for my research and thesis. His detailed comments, enlightening suggestions, and complete corrections help ameliorate this thesis greatly. I have learned much from him.

Next I want to thank Dr. Spencer Smith. Each conversation with him was of great benefit to me. I greatly appreciate his comprehensive review for this thesis. I would like to thank Dr. Emil Sekerinski for reviewing this thesis and being an examiner of thesis defense committee. I also want to thank Dr. John Pryce for his suggestions and help.

Finally, I give my thanks to my parents and my wife for their support and understanding.

Contents

List of Figures

List of Tables

Table 1: Acronyms Used in This Thesis

Acronym	Description
API	Application Programming Interface
ATF	Automated Testing Framework
DAE	Differential-Algebraic Equations
DAETS	Differential-Algebraic Equations by Taylor Series
DDT	Documentation Driven Testing
QA	Quality Assurance
SC	Scientific Computing
SCS	Scientific Computing Software
SDET	Software Developing Engineer in Testing
SE	Software Engineering
USB	Unexecuted Statement Block

Chapter 1

Introduction

Scientific Computing Software (SCS) typically uses finite precision floating-point numbers to represent continuous quantities. It is distinguished from other computer softwares in that it is normally relates to a great amount of mathematical knowledge.

For a long time, domain experts not only take charge of creating mathematical and computational models, and then representing the models into source code of SCS, but also are responsible for verifying their SCS. A question may automatically arise: can normal testers, who are familiar with software testing, can help domain experts improve the quality of SCS? This thesis aims to answer this question by explaining how testers can use the documentation driven testing (DDT) technique and software testing tools to verify and improve the quality of SCS.

This chapter includes four sections. Motivation section (§1.1) explains why SCS development teams need to grasp software engineering (SE) knowledge. Background section (§1.2) defines the terms used in this thesis, describes the current testing work in SCS, and introduces a DAETS — Differential-Algebraic Equations by Taylor Series, which is the case-study used in this thesis. Scope section (§1.3) talks about the testing target of this thesis. Organization section (§1.4) describes how the remaining chapters are structured.

1.1 Motivation

Nowadays, SCS plays an important role in scientific research, engineering and service trade. Experts with domain specific knowledge and scientific computing knowledge develop a large amount of algorithms and libraries, which greatly improve the productivity of the related fields. It is natural to realize that the quality of a SCS decides if the results of SCS can be trusted. To evaluate the quality of SCS one needs to use some evaluating indicators, say, correctness, accuracy, performance, etc.

Among all these indicators, the correctness of SCS is the most important one. A SCS without correctness is useless and dangerous. For example, Oliveira and Stewart [11] present three SCS failures: i) in 1991, the failure of a Patriot missile did not hit an incoming Scud missile; ii) in 1991, the Sleipner A oil rig collapsed; iii) in 1996, the Ariane 5 rocket exploded. All these failures

result in significant losses, and even a tragic loss of human life in the case of the Patriot missile disaster. In addition, the defects of a SCS can cause other developers not to trust the SCS. The software reuse is hampered by the lack of confidence in the code of others.

If the code came with proof that it was extensively verified, this would help build trust. But how? That domain experts develop a SCS normally experience three stages: creating mathematical model for a real world problem; discretizing the mathematical model into a computational model; and implementing it as SCS [4]. Since all three stages rely on approximations, they naturally introduce uncertainties into SCS. To characterize these uncertainties, domain experts need to do code verification, and validation to decide the extent to which the computer implementation corresponds to the computational model, the mathematical model, and the real world problem. However, these domain experts are usually "caretakers" of the models [6]. What they do is to tune a SCS to show the models can work, not to improve the quality of the software. Domain experts are good at science and/or engineering, but they usually lack effective software engineering techniques [6, 7]. As Gregory V. Wilson mentioned, overwhelming domain experts still use plain text editors like notepad, and do not test their programs systematically at all [5]. Therefore, a caretaker of software itself, a tester role, is needed.

The idea presented in this thesis aims to introduce testing methodologies that have been successfully used in SE, so people developing SCS can invite

or directly mimic testers to follow these methodologies for their own work.

1.2 Background

1.2.1 Overview of Verification Methodologies

Both software testing and formal mathematical specification can be used to verify the quality of software. However, the cost of a formal method is normally much higher than that of software testing, and it is generally used for mission-critical project. Hence, software industry usually chooses software testing as verification method to evaluate the quality of software. This thesis emphasizes software testing techniques to verify SCS.

Before explaining how software testing is currently done in software industry, some terms should be introduced first.

`Inspection` — A team of people read or visually inspect a program or a document [3].

`Black box testing` — A test that is based on a component's specified behavior without regard to its implementation [13].

`White box testing` — White box testing assumes that the tester can take a look at the code for an application block and create test cases that look

for any potential failure scenarios. During white box testing, one analyzes the code of the application block and prepares test cases for testing the functionality to ensure that the code is behaving in accordance with the specifications and testing for robustness [13].

`Unit testing` — The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as one expects. Each unit is tested separately before integrating them into modules to test the interfaces between modules. Unit testing has proved its value in that a large percentage of defects are identified during its use [13].

`System testing` — System testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component, and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and expand the process to test modules with those of other groups. Eventually, all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once [13].

`Regression testing` — Any time one modifies an implementation within a program, one should also do regression testing. This can be done by re-

running existing tests against the modified code to determine whether the changes break anything that worked prior to the change, and by writing new tests where necessary. Adequate coverage without wasting time should be a primary consideration when conducting regression tests. One goal is to spend as little time as possible doing regression testing without reducing the probability of detecting new failures in old, already tested code [13].

`Performance testing` — A system test in which you try to demonstrate that an application does not meet certain criteria, such as response time and throughput rates, under certain workloads or configurations. [3]

`Security testing` — A form of system testing whereby you try to compromise the security mechanisms of an application or system. [3]

`Stress testing` — A form of system testing whereby you subject the program to heavy loads or stresses. Heavy stresses are considered peak volumes of data or activity over a short time span. Internet applications, where large numbers of concurrent users can access the applications typically require stress testing [3].

`Equivalence classes` — partition the input domain of a program into a finite number of equivalence classes such that one can reasonably assume (but, of course, not be absolutely sure) that a test of a representative value of each class is equivalent to a test of any other value [3].

Developing software is an engineering activity, which yields an engineering

product called *software*. An engineering product may contain defects, so testers need to verify that its functionality and quality satisfy the requirements of customers. One way to verify the functionality and quality of products is testing, also called quality assurance (QA). As software is also an engineering product, software testing plays an important role in checking software quality. Testing software is mainly performed by testers in the field of SE.

In the current software industry, software testing is executed simultaneously with software developing since SE experience shows that the earlier bugs are found, the lower the cost of fixing them. In the designing phase of a software, testers should design testing specification based on developing specification. When developers begin to implement modules of the software, developers and tester should do unit testing, which belongs to white box testing, for these modules. After the software is completely implemented, testers need to execute system testing, which belongs to black box testing and mainly focuses on the functionality of the software. This testing mechanism can permit the development team and test team to find potential defects of software as early as possible and decrease the cost of fixing them. In addition, the software may be subject to regression testing, performance testing, stress testing, and security testing, etc., based on different testing purposes.

1.2.2 Testing Scientific Computing Software

1.2.2.1 Current State of Testing in SCS

As for most domain experts, testing SCS is based on models. Domain experts usually pay more attention to how much a mathematical model corresponds to its real world problem, but do not pay much attention to the quality of the implementation of SCS [12]. When they tune a SCS, they just want to show that their model works, but may not perform extensive and rigorous testing.

Domain experts, who are aware of the importance of the quality of SCS implementation, introduce to SCS development software testing techniques, which are well known in SE. For example, they carry out unit testing, system testing, regression testing and so on for SCS [4]. However, when designing test cases, testers of SCS can often meet the oracle challenge — in scientific computing (SC), testers rarely know what the true answer to SC problem is, even though most testing methodologies assume that they will have this information. To get out of this jam, communities of scientists interested in SC collect test problems, which are fit for SCS testing, as benchmark test suites. Some problems in the benchmark test suite have known analyzed solutions; thus their solutions can be directly used as the expected results in a test case. Other problems have no known closed-form solutions, but they are still useful for they can be utilized to compare a target SCS with the best solution in the

literature or other competing SCS (also called Parallel Testing). In addition, to verify the derived properties and characteristics of a solution is another way, to overcome this oracle challenge of SCS, for the properties of solution is ofter easier to find [7]. After designing test cases, testers can evaluate the completeness and the ability of test cases by executing code coverage analysis technique (refer to §5.3) and mutation testing technique — insert code faults randomly into source code of SCS and then run test cases to try to find these faults.

In addition, domain experts can also use static inspection technique and static inspection tools to ensure the quality of SCS. This technique requires developers or static inspection tools to review documents and source code of SCS line by line to locate documentation defects, software defects, dead code, infinite loops etc [4].

1.2.2.2 Documentation Driven Testing for SCS

This thesis is dedicated to explain how to inspect the completeness of the documentation i.e. the user guide of public interfaces of SCS implementation and check the correctness of public interface functionality of SCS by mainly using the document-driven testing (DDT) technique [7].

DDT in this thesis is a top-down black box testing technique, and it is usually executed by testers. Hence, this thesis describes the testing technique from a tester's perspective.

Here, the tester mentioned in this thesis is a software development engineer in testing (SDET), which means he ought to be able to write testing scripts to automate testing process, besides grasping software testing techniques. Software testing is intended to find bugs in software systems before users hit them. Testers testing SCS should act as either professional testing engineers or common users, and switch these two roles adeptly depending on the type of the content that needs testing. A tester acting as a professional test engineer knows well the software system that is being tested, so he has the ability to tell the correctness of the contents he wants to test. However, a tester acting as a common user is expected to think or operate the software system like a normal user. A normal user of SCS usually knows little or nothing about the software system. This kind of users need to study how to use the software: they can encounter any type of problems confusing them, and they may make mistakes, sometimes even the mistake that the professional developers and testers can never make. A tester cannot take for granted the assumption that the end users are familiar with the software system and the domain-related mathematical knowledge underlying it. A tester should try to figure out how normal users make use of the software, instead of how developers use it.

When a tester, who has experience of testing business application software, wants to test SCS, he faces a great challenge. This challenge comes from the complexity of mathematical knowledge and particular characteristics of SCS. However, with the help of domain experts and developers, a tester can overcome the challenge and use his testing experience to improve the qual-

ity of the software. In contrast with developers, who own domain specific knowledge and mathematical knowledge, and carry out unit testing, testers use inspection technique to validate the user guide, but do not execute extensively static code inspection for they lack enough experts to help them execute the conventional code inspection, which involves multiple reviews of the code by people that understand it.

1.2.3 Overview of DAETS

This thesis originates from the testing project for testing DAETS: Differential-Algebraic Equations by Taylor Series [1]. DAETS is a software package implemented in C++. It is used to integrate an initial value problem (IVP) for differential-algebraic equation (DAE) system of an arbitrary index and order over a range, using a Taylor series method. DAETS can provide its users with detailed structure information of the DAE and calculate the numerical solution of an IVP either at the end of the range or step-by-step.

John Pryce developed the idea — solving a DAE by Taylor series originated by Y.F.Chang and G.F.Corliss, into a systematic method in 1996. Nedialko S. Nedialkov began to collaborate with John Pryce in developing the theory and code of DAETS in 2002. They offered the first version of the code and user guide in the spring of 2008.

DAETS solves initial value problems for DAE system. The system has the

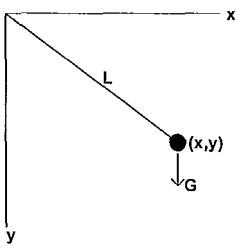
form:

$f_i(\, t,\ the\ x_j\ and\ derivatives\ of\ them\,) = 0, \quad i = 1, \ldots, n,$

in terms of the unknown state variables $x_j(t), \quad j = 1, \ldots, n$. It is defined by a user-supplied function that evaluates the functions f_i [1].

A DAE example: the simple pendulum problem.

Figure 1.1: Simple Pendulum Problem



The simple pendulum system is a DAE of differential index 3, and is defined by the equations:

$$0 = f = x'' + x\lambda$$
$$0 = g = y'' + y\lambda - G$$
$$0 = h = x^2 + y^2 - L^2\ .$$

The gravity G and the length L of the pendulum are constants. The independent variable is time t. The dependent (state) variables are the coordinates $x(t)$, $y(t)$ of the pendulum bob, and the Lagrange multiplier $\lambda(t)$.

For more details about DAETS, refer to the user guide of DAETS [1].

1.3 Scope

The testing methodology described in this thesis is comprised of the user interface testing theory, and some software testing techniques and tools to assist with testing. It aims to help verify and improve the functional requirement i.e. the correctness of the documentation, the public interface functions of code library, and the source code of SCS, but not the non-functional requirement, say, accuracy and performance of SCS.

The core of DDT is to test the user interface of SCS. The user interface in this thesis is composed of the user guide of SCS and the public interfaces of code libraries of SCS. Hence, the user interface is a bridge connecting a software and its users. The user guide is the knowledge interfaces of SCS, and the public interfaces of code libraries are the application programming interface(API) of SCS.

The DDT technique explicitly assumes the existence of a user guide for SCS. The user guide should includes the description of mathematical theory underlying SCS and explanations about how to use the public interface functions of SCS. The depiction of mathematical theory tries to provide users with general information of the theory underlying the SCS. This general information intends to narrow the knowledge gap between users and developers, so users can make a good use of the features of the SCS. The user guide explains detailed information of public interface functions of SCS, such as their pa-

rameters, return values and exceptions they may throw. It also talks about the use experience of these public interface functions, say, good practices, pitfall and traps that users can use or avoid.

The public interface functions of code libraries are the API of SCS. Users of SCS can solve numerical problems by calling these interface functions. If users meet problems in the process of calling these functions, they can consult the description of public interface functions in the user guide.

The software testing technologies involve designing test cases, managing bugs information, and using software testing tools that are available to assist with testing.

1.4 Thesis Outline

The remaining chapters of this thesis are organized as follows. Chapter 2 describes how to verify the user guide of SCS. Chapter 3 discusses about test case design. Chapter 4 involves the reasons and methods of bug management and analysis. Chapter 5 explains how to use software testing tools to help improve the quality and efficiency of SCS testing. Chapter 6 concludes the thesis.

Chapter 2

User Guide Verification

2.1 Purpose of User Guide

A user guide for a software package is a technical document aimed to provide assistance to people using the particular software system.

The user guide needs to summarize the theory underlying SCS. SCS like DAETS is dedicated to solving a particular type of numerical problems, so it uses some domain-related mathematical theory. This kind of mathematical theory is usually abstract and difficult to understand. As a result, a small number of professional mathematicians can develop such kind of software. The users of such packages, however, are normally not familiar with the mathematical ideas. Obviously, there is a huge knowledge gap between

developers and users. The user guide can help users out of this predicament of knowing little about related mathematical knowledge. For example, it can present users with mathematical notions underlying the software, explain hard mathematical ideas in sufficient detail, and sometimes give real examples to decrease the learning curve of the theory. In this way, the user guide provides a bridge to connect developers and users. The user guide needs also to let users know the advantages and disadvantages of this SCS compared to similar ones. It should point out situations that may make the software fail to work properly, or even more subtle cases that make the software does work properly in certain situations.

The user guide is supposed to expose systematically to users the features of SCS and public interface function information, and give enough examples to demonstrate how to use these features and public interface functions. It should also tell users good practices and warn them against pitfalls and traps they may encounter in the process of using the software.

In addition, the user guide should tell user installation information and portability information of the software.

Finally, the user guide ought to be an open system. It should frequently supplement new typical or common questions that users often ask and summarize usage experience based on the interaction between developers and users.

2.2 Verifying a User Guide

A user guide is the knowledge interface between developers and users. It represents what the developers expect users to know. As a user guide directly determines if a user can make full use of the features of the SCS, people can immediately realize the importance of the correctness and completeness of it. In view of its importance, it should be the first target to be verified in the whole testing process.

A good style user guide should cover all the topics that can help the end user employ a SCS correctly and efficiently. Testers can use static inspection technique to verify if the user guide is correct, complete and consistent by checking the information in the following checking list.

Check list for a good style user guide:

1. Summary of expected user characteristics
2. Assumption and limitation of the theory
    (a) Errors and typographical errors
    (b) Understandability and consistency of the explanation of the theory
    (c) Terminology definition-usage order
    (d) Terminology abbreviation and acronyms
3. Description of public interface function

(a) Description of public interface:

i. Parameter list

ii. Return value(s)

iii. Constraints on valid input

iv. Exceptions

(b) Good practices

(c) Pitfalls and traps

(d) Interface information consistence between the user guide and source code

4. Installation and portability information of the software

2.2.1 Expected User Characteristics

Since SCS is usually related to a great amount of theory and math, it is inevitable that the users need to have some knowledge of the basic theory and math underlying the SCS, or the user cannot make good use of the SCS.

For this reason, testers should verify if the user guide explicitly announces the expected characteristics of users. They ought to check if the user guide further provides resources that can help make up the knowledge for the users, who do not grasp the basic theory, so that this kind of users can finally master the ability to use the SCS.

2.2.2 Verifying the Theory Underlying SCS

The mathematical theory underlying a SCS can be the most difficult part of the user guide for both the developer who writes it and the user who reads it. On one hand, the developer cannot explain the theory in great detail due to the lack of space for a detailed description of it. On the other hand, the end user normally knows nothing or little about the background theory of the software system. Under this circumstance, a tester should play his role to check if the explanation of the theory is correct, clear, consistent and understandable for the end user, and to give the developer his feedback to improve explanations, where necessary.

The first step is to read the explanation of the theory carefully to find possible errors and typos in sentences, formulas, and diagrams of the user guide, based on related mathematical knowledge and literature, or assistances from domain experts.

The second step is to verify if the explanation of the theory is understandable and consistent, and if there is a better way to describe the theory. To accomplish this task, a tester is expected to check the design and the organization of the development of the theory, and he is also supposed to ask himself if the meaning of this sentence or this paragraph is clear enough from the point of view of a general user.

Testers should check the correct order of definitions-usage and full name-

abbreviation of key terms and concepts. Since key concepts are important for the description of a theory, and they can frequently appear in the user guide, their definition must appear before their usage. The full names of terms must also appear before their abbreviations in the user guide. Testers need to find those definition-usage and full name-abbreviation that are out of order and then help developers modify them.

In addition to the order of the definition-usage of key concepts, a tester needs to verify if the user guide provides appropriate concrete examples to show how to use the theory to solve real problems. Generally, the description of the theory of SCS is abstract and complex, so it is difficult for the end users to understand completely the theory. To solve this difficulty, the user guide has to provide concrete examples to show end users how real problems are solved. So a tester should make sure if these examples can properly cover abstract concepts users need to grasp.

Finally, testers should pay attention to the complete extent of the description of a method in the user guide. A method usually has its advantages and disadvantages compared to other similar methods. A method may even fail to work in some cases. In this case, the user guide needs to point out problems for which the method is appropriate, and problems for which the method does not work.

2.2.3 Verifying Public Interface Description

Public interface functions is the application programming interface (API) of SCS. The users of SCS perform their computing task by calling these functions. Therefore, the description of these functions should be clear, complete and helpful.

When testers verify the descriptions of public interfaces in the user guide, they should first check if each public interface function has its corresponding description document. Then, for each specific function, testers are supposed to examine if the documentation of the function elaborately describes its parameter list, return value(s), constraints on valid input and exceptions the function may throw and reasons that lead to them.

In addition, testers should check if the description provides good practices, pitfalls and trap of these functions. Users may encounter many problems in the process of calling public interface functions of SCS. Some problems originate from users lacking experience, and others may result from the features of language or design defects. Good practices can help user efficiently call these functions, and the information of pitfalls and traps can help users avoid problems.

Finally, testers need to verify the consistency of public interface functions between source code and the user guide. The consistency means public interface functions in source code and user guide should have identical parameters lists,

return values and exceptions lists (The order of the parameters in the list must also be the same).

2.2.4 Verifying Installation and Portability Information

The installation of a software is the first step of user's experience. A user's first impression about the software may be influenced by whether the description of the installation steps is clear and unambiguous, and whether the installation of the software, based on the installation steps, is successful. Given this, a tester should carefully verify the usability of installation information. To verify the usability of installation information, a tester should check both the completeness of installation steps and support information for a software installing. As for the support information, the user guide should describe how the software can be obtained, say, downloaded from a website, and show the supported platforms for this software, say Linux, Mac or Windows. The user guide also needs to provide email addresses or web sites for users to get further support information from developers.

For the installation steps, the user guide should first list all the dependent libraries and third-party components that are necessary to install the main software. For each dependent library or component, the document ought to tell users where to find this component, and how to install it. The user guide needs to show how to install the main software based on all dependent components. After that, the user guide should provide some way to verify

if the installation is successful. For example, if the software package is a library, the user guide can provide an application program to call this library and its expected result.

The installation information should also include a section about frequent asked questions, which lists the most common problems users may encounter when installing the software and the methods to solve these problems.

After verifying the completeness of the installation document, the tester should try to install the software by following the installing information step by step and make sure the software can be indeed installed successfully on all the supported platform.

Finally, the user guide needs to provide the portability information of the software. It should point out clearly which kind of platforms can support this software, and tell users the difference of installation process in different platform.

2.2.5 Results from Inspecting the User Guide of DAETS

By utilizing static inspection technique to verify the user guide of DAETS, we find 7 document issues.

1. Some issues result from mathematical mistakes and typos.

   For example:

(a) Mathematical mistakes:

It was discovered that the formula for the simple pendulum system should be

$$\begin{pmatrix} f \\ g \\ h \end{pmatrix} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} x'' \\ y'' \\ \lambda \end{pmatrix} + \begin{pmatrix} 0 \\ -G \\ x^2 + y^2 - L^2 \end{pmatrix},$$

instead of

$$\begin{pmatrix} f \\ g \\ h \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} x'' \\ y'' \\ \lambda \end{pmatrix} + \begin{pmatrix} x\lambda \\ y\lambda - G \\ x^2 + y^2 - L^2 \end{pmatrix}.$$

(b) Typos:

Another example is that one of the public interface declarations should be "`int getOrder()const`", but was written as "`void getOrder() const`".

2. Other issues are related to software design problems.

For instance, we find the interface function `void printDAEinfo() const` is designed to output information only to the monitor. However, the function with an `ostream` parameter, like

```
void printDAEinfo(ostream &s = cout)const,
```

can provide users with more choices, say, outputting into a file, to a string, or just to the screen. This software issue was corrected based on the advice of the author.

It is obvious that static inspection for the user guide improve not only the quality of document, but also the usability and testability of the software.

Chapter 3

Test Case Design

Software may contain defects like other engineering products, thus software testing is a necessary step in the process of developing software. Many different testing techniques can be used to verify the functionality and quality of a software, depending on different testing targets, intentions and operators. This thesis results from the testing project of the DAETS package, and this project focuses on the verification of the user interface of DAETS by using DDT, which mainly belongs to black box testing. In addition, code coverage testing — one of the white boxing testing methods, is used to supplement DDT to improve the quality of DAETS further.

Designing test cases is one of the core tasks of testers. The purpose of designing test cases is to organize properly the testing activities to find as many defects as possible. However, both testers and developers should recognize

that test cases can only prove the existence of software defects, but can never prove the absence of software defects.

3.1 Designing a Template for Test Cases

Before designing test cases, testers need to create a template for test cases. This template specifies what kind of information should be presented in each test case. Because designing test cases usually takes professional testers familiar with domain knowledge or testing skills plenty of time, test cases should not be discarded easily (unless the software they test is not used anymore). Test cases ought to be a memo, carefully recording how those professional testers have tested the software. With the information in these test cases, any tester should be able to reproduce the testing process. Hence test cases can be repeatedly executed in the same high testing quality throughout the software developing process. To achieve this goal, a guideline is needed to decide what kinds of information in test cases can lead to a repeatable and consistent testing quality. This guideline is the template for test cases.

To design a test case template, a tester needs to imagine how a tester, who is new to the software system, can succeed in executing test cases only with the descriptions of test cases. Hence, a test case should have a name to tell testers which one needs to be executed. A test case needs to have a description of its purpose, so testers know what this test case is intended to do. A

test case should provide input data or input scenarios for public interface functions of the software. It needs an expected result to be compared with the actual result returned from a function. The compared result can be used to tell if the test case executes successfully. A test case also needs to provide detailed steps to guide testers how to execute this test. At last, a test case may have the traceability between a test case and its origin, so the test case is connected with the user guide and the public interfaces of source code, etc. This traceability can help testers make sure test cases cover all the functionality of the software. In addition, it can verify the consistency among test cases, user guide and the public interface functions of the source code. Finally, it can help testers and developers quickly locate and correct errors in the user guide and source code, when a related test case fails.

To summarize the template for test cases should include the following information:

1. Test case name
2. Purpose
3. Detailed reproduction steps
    (a) input data or input scenarios
    (b) steps in details
    (c) expected result
4. Traceability between a test case and its design origin

An example follows:

1. Test case name:

   `DAEsolutionTest::testCtrThrowLogicErrorException`

2. Purpose:

   To test if the constructor of the `DAEsolution` object [1] throws `std::logic_error` exception

3. Reproduction steps:

   (a) Create a `DAEsolver` object [2] with an ill-posed `sigmaMatrix`[3]

   (b) Call `DAEsolution` constructor with the just created `DAEsolver` object as its parameter

   (c) The `DAEsolution` constructor should throw a `std::logic_error` exception

4. Traceability:

   This test case originates from the description of the constructor of `DAEsolution` class in DAETS user guide, §1.3.3.

After designing the template of test case, testers can begin to design test cases for public interfaces and user scenarios.

[1] refer to §1.3.3 of [1]
[2] refer to §1.3.2 of [1]
[3] refer to §5.1 of [1]

3.2 Designing Test Cases for Public Interfaces

To design test cases for a public interface is not a simple, one-step task, but a task containing a series of steps. In this design process, the verification of the user guide and the design and execution of test cases are interwoven. Black box testing and white box testing are used in succession, to make test cases as complete as possible. When test cases fail, testers should try to figure out the problems and help developers correct the related source code and user guide. The detailed design process is described step by step in this section.

3.2.1 Black Box Method for Designing Test Cases

After a tester verifies the consistency of public interfaces, he needs to design test cases based on the description of public interfaces in the user guide. This method belongs to black-box testing. The designing process of test cases can be divided into three steps:

1. Verifying the descriptions of public interfaces in the user guide
2. Designing test cases
3. Checking the completeness of test cases

The first step is to check if the description of the public interfaces in the user guide are understandable and helpful.

Understandable descriptions means the explanation of the public interfaces is proper and clear, and helpful descriptions means they provide good practices, and pitfalls and traps about these public interfaces. Understandable and helpful descriptions are important to both testers and users. For testers, understandable and helpful interface descriptions can be efficiently converted into test cases for covering all the possible types of input data and exceptions. In addition, testers need also realize that vague interface descriptions often imply design problems of public interfaces. For users, understandable and helpful interface information can help them correctly and efficiently call these public interfaces without errors.

The second step is to design test cases based on the descriptions of public interfaces in the user guide. Testers are expected to create test cases for each parameter, and the exceptions that each invalid parameter can trigger by using the ideas of equivalence classes [3] and boundary value analysis [3].

For each parameter of a public interface function, there must be a detailed description of its input range in the user guide. This input range should be divided into two groups: valid input value group and invalid input value group. These two value groups are called equivalence classes, and the operation of dividing these two groups is called equivalence partitioning. Sometimes each of these groups can be divided further into several disjoint sets. After equivalence classes are created, boundary value analysis can be used to cover test cases in which their input values lie on the boundary of each equivalence

classes and also below and above the boundary.

Example. The function `getX` [1] of `DAEsolution` class is described in the user guide of DAETS as follows:

```
double getX (int index, int order) const throw(std::logic_error, std::out_of_range)

Input:  index, order.  Such that x.getX(j-1,k) returns the current value of the entry of
x representing x_j^(k), the kth derivative of the jth variable.

Constraint:  (j,k) must be in the index set J. Otherwise, an exception std::out_of_range
results.  getX must be called on initialized entries.
If getType(index,order) == Uninitialized, an exception std::logic_error results.

Returns:  x.getX(j-1,k) returns the current value of the entry of x representing x_j^(k) .
```

Given that a tester creates a `DAEsolver` object and a `DAEsolution` object for the simple pendulum problem, the range of the parameter index is [0,1] and the range of the parameter order is [0,1].

When the tester begins to design test cases, he should first partition equivalence classes for the parameters index and order. Based on the interface description information in the user guide and this simple pendulum problem, the tester knows the ranges of index and order. For the parameter index, it can be divided into two equivalence classes, one valid input group including the values 0 and 1, and one invalid input group including the two disjoint sets, whose values are below 0 and above 1. Then the tester should continue to do boundary value analysis for the parameter index and get the result that all the possible input values for index can be picked up from four sets:

{all the input values less than 0}, {0}, {1}, {all the input values larger than 1}. And the analysis process for the parameter order likewise.

Then the tester should also design test cases to cover the exceptions, which may occur when calling this function. There are two possible exception for `getX` function: `std::logic_error` and `std::out_of_range`. For the exception `std::out_of_range`, the tester knows from the interface description that the valid input group of index and order will not cause `getX` to throw `std::out_of_range` exceptions and that the invalid input group of index and order must cause the function to throw `std::out_of_range` exception. For the exception `std::logic_error`, the tester can create a `DAEsolution` object with initial variable values, then call the function `getX` to verify if the function does not throw the exception `std::logic_error`. Then the tester should create a `DAEsolution` object with uninitialized variable, and call `getX` to verify if `getX` does throw the exception `std::logic_error`.

Table 3.1: An Example of Test Case Design

index and order	variables	correct	out_of_range	logic_error
index $\in$ [0,1], order $\in$ [0,1]	initialized	X		
index $<$ 0, order $\in$ [0,1]	initialized		X	
index $>$ 1, order $\in$ [0,1]	initialized		X	
index $\in$ [0,1], order $<$ 0	initialized		X	
index $\in$ [0,1], order $>$ 1	initialized		X	
index $\in$ [0,1], order $\in$ [0,1]	uninitialized			X
index $=$ 0, order $\in$ [0,1]	initialized	X		
index $=$ 1, order $\in$ [0,1]	initialized	X		
index $\in$ [0,1], order $=$ 0	initialized	X		
index $\in$ [0,1], order $=$ 1	initialized	X		

Based on this black box analysis process, the tester can find all the possible test cases covering cases described by public interface information in the user guide.

The third step is to verify the completeness of test cases for each public interface function by checking the following list:

1. A test case with all parameters on valid input range
2. For each parameter
   (a) for each exception the parameter can trigger, a test case with the parameter on invalid input range (above the correct input range), other parameters on valid input range
   (b) for each exception the parameter can trigger, a test case with the parameter on invalid input range (below the correct input range), other parameters on valid input range
   (c) for each exception the parameter can trigger, a test case with the parameter on valid boundary input value, other parameters on valid input range
3. If there exist other descriptions of the public interface function in the user guide that can be translated into test cases

Testers can create the complete traceability between the user guide and test cases by following this list.

3.2.2 White Box Method for Designing Test Cases

If a tester wants to find all the defects in a software by using the black box method, he must do the exhaustive input testing in the entire input domain. However, this normally turns out to be impracticable due to the complexity of input domains. In this case, the tester may have to supplement equivalence classes analysis and boundary value analysis with the white box testing method.

Statement coverage of source code is often used in white-box testing. This method expects every statement in the program to execute at least once. Although this method is not as strong as other complex white box methods, like condition coverage, it does work to supply supplementary test cases to find defects, which cannot be spotted by the black box testing.

Data for statement coverage can be obtained by executing code coverage testing. With such coverage data, a tester can easily locate unexecuted statement blocks. This kind of unexecuted statement block (USB) is the aggregates of a sequence of unexecuted statements, which do not have branch and jump statements. For each USB, the tester has to analyze the calling chains from the function, where this USB is located, until those public interface functions, so as to trace a reversed path from the public interface function to this function. Then the tester may try to find the appropriate input values to trigger these calling chains. If such input values can be discovered, the tester is able to create a new test case to make the execution flow to hit this

unexecuted statement at last. However, sometimes the tester cannot find proper input values for covering this unexecuted statement block, and this often implies that some defects may exist somewhere in these calling chains.

More details about how to use code coverage tools are described in §5.3.

3.3 Designing Test Cases for User Scenarios

The test cases designed by the black box testing mentioned in the last section can only test one public interface function at a time. However, some program defects may occur only when the business logic of multiple public interface functions communicate with each other. For this kind of defects in the program, there is nothing the test cases designed in the last section can do to detect them.

In this case, the concept user scenario is introduced. A user scenario is a series of interaction between a user and a software system to make the software system accomplish a certain task. This series of interactions between the user and the software system must involve calling multiple public interface functions. Normally, these user scenarios can appear in the user guide as examples that teach users how to use the features of the software package to solve real problems.

Testers can use these scenarios to design test cases. These test cases are

called system integration test cases. Operations of this kind of scenarios are described as a series of public interface function calls. Hence an integration test case can be designed as these public interface function calls. The execution of the test case can also return a result. This returned result should be compared with the expected result supplied by the scenario in the user guide. The compared result is used to tell whether the system integration testing succeeds.

3.4 Results of Black Box Testing

By executing test cases designed by black box testing technique, we found 18 software implementation defects and software design defects on the DAETS library.

The first class issues are related to implementation errors of source code. For instance, a test case discovered that the function

```
void getCVector( vector<int> &c) const
```

did not returns "C" vector, but "D" vector [1]. By checking the source code of the function, we located the error in the function. The defect is corrected by making the function return the correct "C" vector.

The second class issues result from design problems. For example, a test case found that the function

```
void getSigmaMatrix(vector< vector<int> > &s, int neginfval = -1)
```

can assign a positive number to the parameter `neginfval`, which appear in sigma matrix to represent an infinite value. However, the positive number can lead to an ambiguous meaning in sigma matrix. It is the design defect that cause this issue. To avoid this problem, some code was added into this function to forbid positive numbers from being assigned to the parameter `neginfval`.

The third class issues originate in the features of programming language. For the public interface function

```
DAEsolution & setX(int index, int order, double value, VarType type = Free)
```

the type of the fourth parameter `VarType` is an enumerate type. In C++, enum type argument can be assigned to an integer or a double type parameter without triggering exceptions. However, this feature of C++ can cause users to make mistakes in some situations. For example, users may call the function with only two integer arguments and one enum type argument by mistake, and it would lead to a strange result other than what the users expects. This is because the enum value argument is assigned to the double type parameter. A test case mimicking the previous calling found this issue. The warning information was added into the descriptions of the function in the user guide, to prevent user from making such mistake.

The fourth class issues are about multi-platform problems. A test case was designed to verify the function `void setHmax(double hmax)` by assigning a double value to the parameter `hmax`, and the test case expects the function throws

a `std::logic_error` exception. The test case did throw the exception on Mac OS, but led to Segmentation fault in Ubuntu. The function was modified to let the test case pass on different platforms.

Chapter 4

Bug Management and Analysis

4.1 Bug Management

When testers verify the user guide and design, implement and execute test cases, they may find document issues, defects in the source code, software design problems etc. To resolve effectively these defects and issues, testers must completely collect their information and trace their fixing process.

4.1.1 Bug Template

Just like the purpose of the template for test cases, testers also need a bug template to instruct them how to record a bug effectively.

What kind of information about a bug can help testers and developers trace and fix this bug? Recording a bug needs an *ID* item and an *Index* item to represent a source file or a document, where the bug is found. *Bug Description* item describes what the bug is, and *Reproduction Steps* item tells how to reproduce this bug. *Status* item means the status of the bug, say, not fixed, partially fixed or fixed. *Bug type* item categorizes the type of the bug. *History* item traces the process of fixing the bug. *Open by* item points out who finds this bug, and this item is useful for a test team.

The bug template is summarized as follows:

1. ID
2. Index
3. Bug Description
4. Reproduction Steps
5. Status
6. Bug Type
7. History

An example follows:

1. ID: 3

2. Index: `DAESolver.h`

3. Bug Description:

   void getSigMatrix(Vector< vector<int> >) &s, int neginfval=-1)
   const throw (std::logic_error); in DAESolver.h, line 31.
   The second parameter of getSigMatrix function can accept a
   positive integer number and 0, however, this parameter can
   only accept a negative integer number based on the information
   in the user guide.

4. Reproduction Steps:

   1) create a solver object for SimplePendulum problem
   2) input a correct vector S with 3*3 structure
   3) call getSigMatrix function
   4) output the signature matrix from the vector S

5. Status: fixed

6. Bug Type: software design problem

7. History:

   (a) Tester:

   Bug ID3 opened.
   I give a positive number to the parameter neginfval, but
   the function accepts it. I want to know if the function
   should refuse to accept the positive number or 0 as

the argument by throwing a logic error exception. 08/07/30

(b) Developer:

See what the user guide says and suggest changes,

if necessary. 08/07/31

(c) Tester:

I feel source code should prevent people from inputting

positive number and 0 once what he input is the same

positive value or 0 that structural analysis will create.

I think the parameter name "neginfval" cannot prevent users

from inputting positive value. The User Guide just prompts

users to use -1 to represent negative infinite, if we should

modify the User Guide to inform users not to input positive

number. 08/08/01

(d) Developer:

The function now prevents users from inputting positive

numbers and 0 by throwing an exception logic_error and

the user guide is updated for this function. 08/08/05

(e) Tester:

Fix verified and closed the bug 08/08/05

4.1.2 Bug Management Tool

In the process of developing of SCS, if testers have no dedicated software to manage bugs, they can use spreadsheets, say Excel or the spreadsheets of Google docs, to save bugs as below.

ID	Index	Bug Description	Repro Steps	Status	Bug Type	History

The spreadsheet used as a bug depot must have several features. A spreadsheet first must be the only one copy and be shared by all the testers and developers. In other words, all the testers and developers access and modify the same spreadsheet file, and so they can always see the real time update of the file. The spreadsheet must also support its users to sort the information. This sorting feature can help testers and developers categorize bugs so that testers and developers can efficiently summarize, analyze and fix bugs.

In DAETS testing project, the spreadsheets of Google docs is used as management tool.

4.2 Bug Handling

Once the design of test cases is complete, testers need run them to verify if functions can work properly. If a test case fails to work, testers have to record the failed test case based on the instructions of the bug template and constantly update the test case by tracing the handling process of the bug.

However, a good tester should not be satisfied with only finding bugs in a program. A good tester is always providing feasible solutions for developers to fix these bugs in the source code. He is always trying to give users a better user guide with good practices about how to use public interface functions or pitfalls and traps that users may encounter when calling public interface functions. With the detailed information of a recorded bug, testers should first locate the defects and analyze the reason resulting in this failure by checking the user guide and examining source code. After that a tester should figure out feasible solutions to help developers fix this bug. He should also try to revise the user guide to tell users how to make good use of public interface functions and how to avoid the pitfalls and traps of public interface functions.

After fixing these bugs, testers need to do more searching in the places where bugs happen. According to the software testing experience of Microsoft, about 80% of bugs are located in 20% source code areas (This rule is a variant of the Pareto principle) [14]. This means the probability of the existence of more bugs in a section of a software is proportional to the number of bugs already found in that section. In addition, when a developer fixes a bug, this fixing may introduces more bugs for this feature. In this case, testers should try more similar input values in the input domain resulting in this bug before and after fixing a bug. If new bugs are found in this way, testers must add new test cases dedicated to cover these defects in the future regression testing, because old test cases cannot find these defects.

4.3 Bug Categorizing and Analyzing

Fixing bugs does not mean a tester's work is complete. Testers should continue to organize and categorize all the defects and bugs found in the testing process, based on different demands to partition groups. For example, those bugs can be categorized by the files in which they belong to, or they can be partitioned by those types say document issue, software design issue, usability and testability etc. The bug management tools can help testers improve their work efficiency.

After categorizing bugs, testers should try to find common defect types and their reasons for each group of bugs to get the relevant experience. Testers should also pay more attention to these common defect types and reasons in the next version of the software by designing some corresponding test cases to cover these defect types. In addition, these common occurred defects often imply the weakness of a developer's programming style. Testers should provide developers with these data to help them improve their programming style.

Chapter 5

Software Testing Tools

5.1 Introduction

Software testing tools can execute many tedious and repeated tasks, analyze source code and manage the defects found in the testing process. These software testing tools can improve testers' production capability and help produce a better quality software.

This chapter covers automated test framework (ATF) and code coverage analysis tools.

5.2 Automated Test Framework

5.2.1 Introduction

ATF is a testing system, which is used to automate the Unit Testing, System Testing and Regression Testing in the process of developing software.

ATF introduces test cases designed for reusability. Test cases are normally designed by professional testers, who are familiar with the features of the software, so they should not be thrown away, unless the software they test has no value any longer. For this reason, ATF acts as a container holding test cases. With this set of test cases, the framework can repeatedly execute the testing of a software, as if each time it is a professional tester, but not a novice who performs the testing. This means the test framework can always provide consistent test coverage and consistent test quality. It is obvious that ATF is very suitable to be used in regression testing.

ATF can also organize and schedule test cases. In a testing framework, all the introduced test cases are organized into several test suites. Each test suite is a collection of test cases, which is used to test dedicatedly some specific feature of the software. Thus, different test suites can be scheduled by the testing framework to test different features of the software. This feature of a testing framework give testers flexibility to choose freely the corresponding test suites to execute based on the test target.

In addition, ATF owns facilities to provide common test environment for test cases. The feature lets testers focus on designing and implementing the real test logic of test cases of a software. ATF can automatically run test cases against an application. It monitors the running, verifies the expected and actual results and reports defects in real time. At the end, it provides detailed statistical data of test results in several optional output styles.

5.2.2 Automated Test Framework CppUnit

For the merits mention above, we introduces the ATF CppUnit, into the testing project of DAETS, to automate the testing process, especially the regression testing.

5.2.2.1 What is CppUnit

CppUnit [2, 9] is a port of JUnit, which belongs to the well-known xUnit testing family [2]. It is implemented in C++, and it is used to test C and C++ programs. Its basic architecture and usage closely follow the xUnit model. CppUnit reduces test cases design and implementation overhead by providing consistent executing environment for test cases. It enables reuse and grouping of test cases to improve efficiency and scalability of the testing. It automatically runs and monitors the selected test cases or test suites, and

collects and presents the test results to test engineers with such different types output format as standard output, plain file or XML file.

Since it is an automated test architecture, which owns all the features described in the introduction subsection, CppUnit is used as the automated test framework in the testing project of DAETS.

5.2.2.2 Architecture of the Test Suite for Testing DAETS

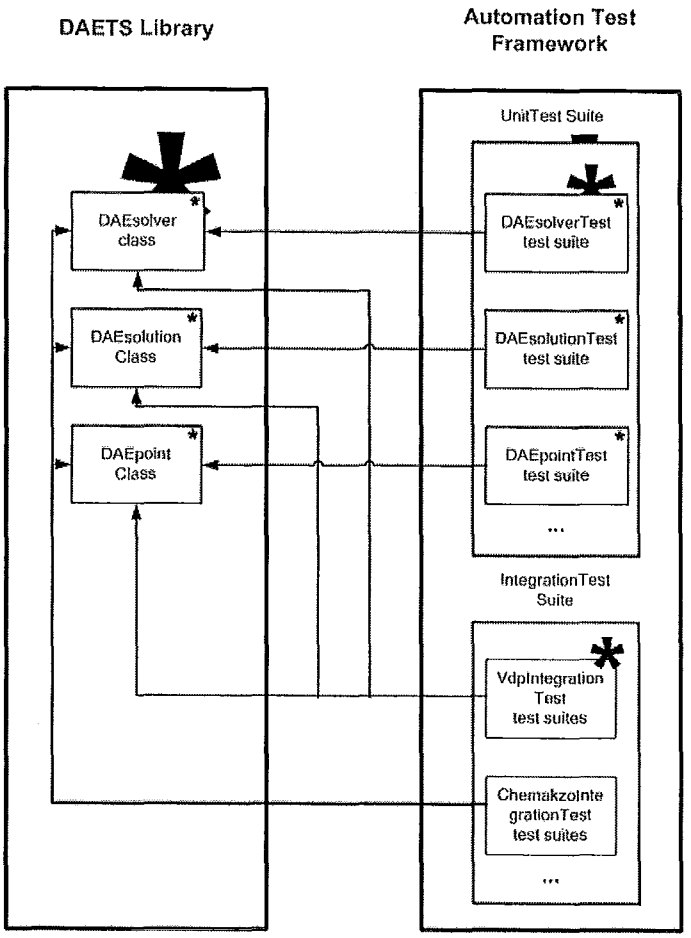
The whole DAETS testing project is organized as a hierarchical structure, which includes test suites and test suite groups. A test suite is used to contain test cases, provide the common testing environment for test cases and register test cases into the CppUnit test framework. A test suite group is responsible for organizing test suites and helping CppUnit test framework select appropriate test suites to execute.

1. Organization of a Test Suite

   The first level is the test suite level. In the DAETS testing project, there are two different type of test classes. One type of test classes only takes charge of testing a single class of source code, and the other type of test classes is responsible for testing multiple classes of source code simultaneously. Each test class mentioned above is registered as a corresponding test suite in CppUnit.

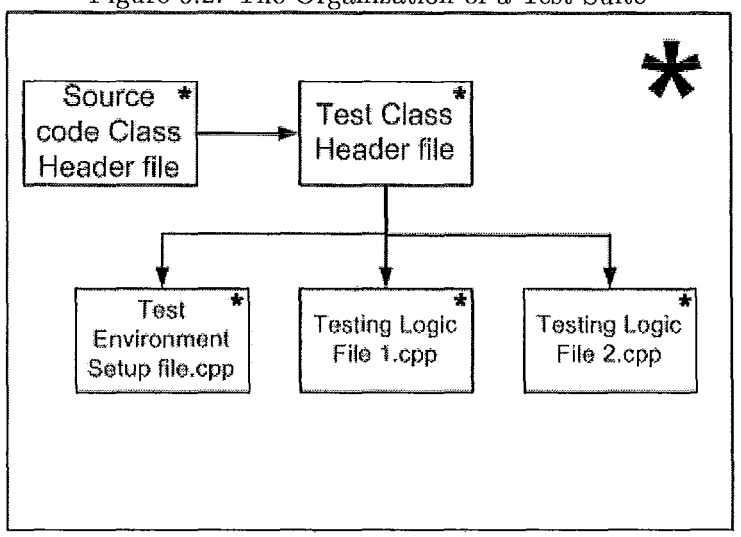
Figure 5.1: The Relationship between DAETS Library and ATF

The stars in the figure below are automatically created by Visio [15].



DAEsolver in DAETS library box holds the integrate function, DAEsolution holds integration result and DAEpoint holds the structure of integration result. UnitTest suite in ATF box includes the test suites DAEsolverTest, DAEsolutionTest, and DAEpointTest, etc. Each of them corresponds to a unit test class, e.g. DAEsolverTest suite corresponds to DAEsolverTest class to test DAEsolver class. IntegrationTest suite consists of VdpIntegrationTest and ChemakzoIntegrationTest etc. Each of them corresponds to an integration test class. VdpIntegrationTest is used to test Van Der Pol problem. ChemakzoIntegrationTest is used to test Chemical Akzo Nobel problem.

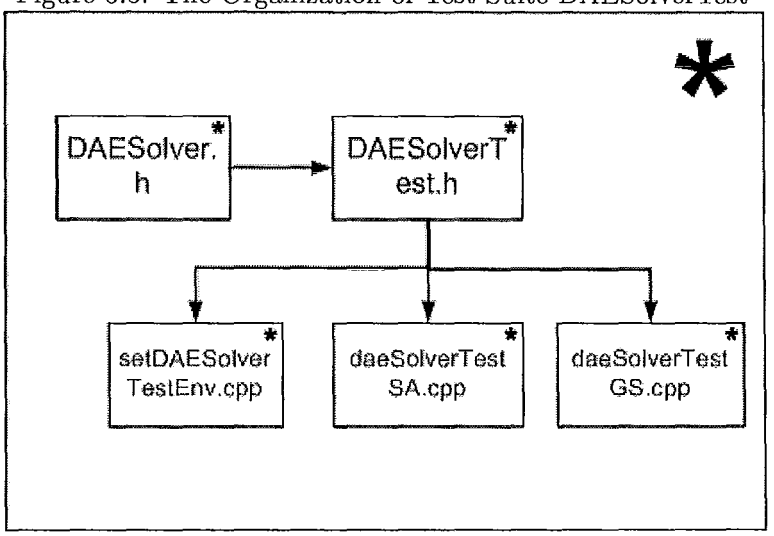
Figure 5.2: The Organization of a Test Suite



A test class normally consists of a test class header file, which includes the test target class header file, a test class environment configuration file, and two or more testing logic files, which are used to store test cases. All test cases in a test suite are designed to test the public interface functions of its corresponding source code class. To simplify the maintenance of the testing project, test cases in a test suite are categorized into two or more different testing logic files based on their testing purposes.

A test class is usually named after its corresponding source code class's name plus the suffix name Test. This is a good practice to create the

Figure 5.3: The Organization of Test Suite DAESolverTest



relationship between testing code and source code. Here, `DAEsolverTest` test class is used to explain the organization of a real test suite.

As the main testing method of DAETS testing project is black box testing, the public function interfaces of DAETS package are the main testing targets. `DAEsolverTest` test class aims to test the public interfaces of the `DAEsolver` class. It is composed of two testing logic files `daeSolverTestSA.cpp` and `daeSolverTestGS.cpp`, the testing environment configuration file `setDAESolverTestEnv.cpp`, and the testing class header file `DAESolverTest.h`.

(a) Testing logic file

By observing the public interface of the header file of source code class `DAEsolver`, testers can notice that these public functions in `DAEsolver` class can be categorized into two groups. One group contains the functions to report the structural analysis data when analyzing a DAE [1], and the other group is composed of the functions to set and get parameters for an integration process and the integrate function itself. For the sake of separating the testing functionality, simplifying management and improving maintainability, the two testing logic files `daeSolverTestGS.cpp` and `daeSolverTestSA.cpp` are created to test their corresponding features of the `DAEsolver` class. These two testing logic files are actually test case container files. The test cases designed to test the public interface functions of source code class are implemented as test methods and saved into testing logic files.

(b) Testing environment configuration file

From `daeSolverTestGS.cpp` and `daeSolverTestSA.cpp` testing logic files, testers can find that most test cases use the common test environment facilities. For example, test cases use a common `DAEsolver` object implemented from the same problem domain DAE function. For convenience, the test environment configuration file is responsible for creating and deleting these common testing en-

vironment facilities for each test method needing them, by using functions `setUp` and `tearDown`, which are class methods in CppUnit. In the process of executing those test methods, the function `setUp()` is first called to initialize the testing environment, before each test method begin to execute, and `tearDown()` is finally called to clear up the test environment after each test case finishes.

(c) Testing class header file

The header file `DAESolverTest.h` is the interface declaration file of the testing class `DAEsolverTest`. It declares all the test methods implemented in testing logic files and all the auxiliary environment setup methods in the testing environment configuration file. In addition, the header file also uses CppUnit's Macro definition to register all the testing methods into the corresponding test suite i.e. CppUnit test framework. With this registration information, CppUnit can retrieve test methods, execute them and collect their results.

2. Test suite group — Test suite organization, selection and execution

The header file of a test class, testing environment configuration file and testing logic files comprise a complete test class. Each test class is

registered as a test suite of CppUnit automated test framework in the header file of this test class. After implementing these test classes and then registering them as test suites, testers need to organize further these test suites as test groups.

(a) Test suite organization

CppUnit uses a hierarchical structure to organize and manage test suites. It has a default suite called ***all*** which is a universal test suite. Two child suites `UnitTest` and `IntegrationTest` are created by CppUnit macros, say, `CPPUNIT_REGISTRY_ADD_TO_DEFAULT( "UnitTest")` is used to register the child test suite `UnitTest` into ***all*** suite.

From what is mentioned in the beginning of this subsection, test classes are divided into two groups. One group is used to test the public interface of a single source code class, and the other is used to test the public interfaces of multiple source code classes. In turn, all the test suites for testing single source code class are registered into `UnitTest` test suite, and all the test suites for testing multiple source code classes are registered into `IntegrationTest` test suite. For example,

```
CPPUNIT_REGISTRY_ADD( daetsTest::DAEsolverTest::getSuiteName(),
"UnitTest")
```

is used to register `DAEsolverTest` suite into `UnitTest` test suite.

In this way, the whole family of test suites becomes a test suite tree. The test suite ***all*** is the root node, and it has two children nodes `UnitTest` suite and `IntegrationTest` suite. Each child suite has several children test suites. The header file `daeTest.h` is used to create this tree-like hierarchical structure.

(b) Test suite selection and execution

After the tree-like test suite is successfully created, a single test suite, or a test suite group say `UnitTest` or `IntegrationTest`, or the default test suite ***all*** can be selected by its suite name and then executed by CppUnit test framework in the file `DAETest.cpp`.

For instance, the following three statements select `DAEsolverTest` test suite, `UnitTest` test suite, and all the test suites respectively.

```
CPPUNIT_NS::Test *suite =
    CPPUNIT_NS::TestFactoryRegistry::
        getRegistry(DAEsolverTest::getSuiteName())
            .makeTest();
CPPUNIT_NS::Test *suite =
    CPPUNIT_NS::TestFactoryRegistry::
        getRegistry("UnitTest").makeTest();
CPPUNIT_NS::Test *suite =
    CPPUNIT_NS::TestFactoryRegistry::getRegistry()
        .makeTest();
```

Besides selecting test suites, the file `DAETest.cpp` also takes charge of executing, monitoring the selected test suites, and collecting and presenting the test results to test engineers with several optional types of output.

For more details about the organization, selection and execution of test suites, refer to Appendix A.

5.2.3 How to Add a Test Suite into DAETS Testing Project

Here, the test suite `DAEsolverTest` is used to show how to add a test suite into DAETS testing project.

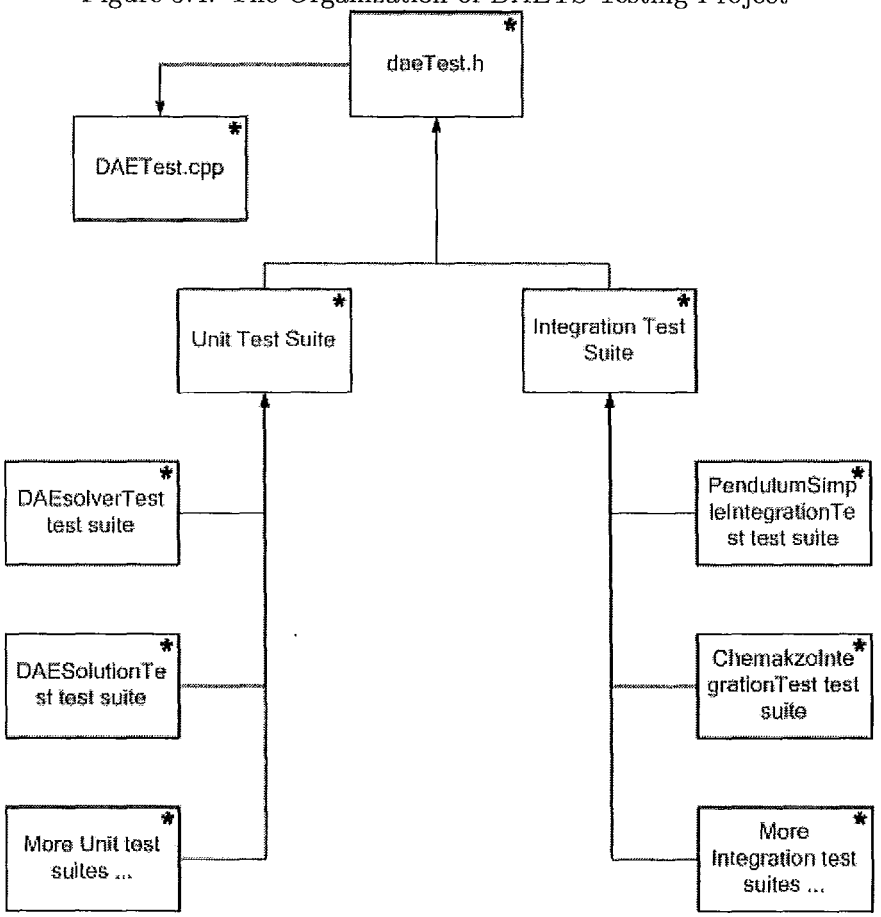
5.2.3.1 Creating a Header File for the Test Suite

The first step is to create a header file `daeSolverTest.h` for `DAEsolverTest` test class as follows.

Test class `DAEsolverTest` needs registering first as a test suite by macros `CPPUNIT_TEST_SUITE` and `CPPUNIT_TEST_SUITE_END`.

Test methods `testGetCVector` and `testSetHmaxThrow` are two test cases implemented in testing logic files. These two methods need declaring in this header

Figure 5.4: The Organization of DAETS Testing Project



file below the public keyword. They also need registering into the test suite DAEsolverTest by CPPUNIT_TEST and CPPUNIT_TEST_EXCEPTION.

The auxiliary functions setUp, tearDown and getSuiteName are public members, and they are used to setup the common executing environment, and data members used by these functions are private members.

The snippet of daeSolverTest.h is:

```
#ifndef SOLVERTEST_H
#define SOLVERTEST_H
#include <cppunit/extensions/HelperMacros.h>
#include DAEsolver.h
namespace daetsTest
{
class DAEsolverTest :public CPPUNIT_NS::TestFixture
{
        CPPUNIT_TEST_SUITE( DAEsolverTest );
        CPPUNIT_TEST( testGetCVector );
        CPPUNIT_TEST_EXCEPTION( testSetHmaxThrow,
                                std::logic_error );
        CPPUNIT_TEST_SUITE_END();
public:
        void setUp(void);
        void tearDown(void);
        static std::string getSuiteName(void);
        void testGetCVector(void);
        void testSetHmaxThrow(void);
```

```
private:
        daets::DAEsolver *ptrSolver1;
        double t0;
        double tend;
        int n1;
};
}
#endif
```

5.2.3.2 Creating Environment Setup File for the Test Suite

The auxiliary functions`fcn1`, `setUp`, `tearDown` and `getSuiteName` are implemented in environment setup file `setDAESolverTestEnv.cpp`.

The function `setUp` create a common executing environment for all the test cases. it provides the start-point `t0` and end-point `tend` of the solution path of Differential-Algebraic Equations (DAE), points out the number of the equations of DAE `n1`, and create a solver object to analyze the DAE problem. The function `tearDown` cleans the common executing environment by destroying the solver object that `setUp` creates. The function `getSuiteName` is used to register the test class `DAEsolverTest` as a test suite by returning the name of the test class. Finally, the function `fcn1` represents DAE equations of the simple pendulum system. It has three equations representing respectively:

$$f = x'' + \lambda x$$

$$g = y'' + \lambda y - G$$

$$h = x^2 + y^2 - L$$

The snippet of setDAESolverTestEnv.cpp is:

```
#include "daeSolverTest.h"
using namespace std;
using namespace daets;
namespace daetsTest
{
/* The DAE functions below come from pendulumsimple.cc well-
   posed */
template <typename T>
static void fcn1(T t, const T *z, T *f, void *param)
{
    // z[0], z[1], z[2] are x, y, lambda.
    const double G = 9.8, L = 10.0;
    f[0] = Diff(z[0],2) + z[0]*z[2];
    f[1] = Diff(z[1],2) + z[1]*z[2] - G;
    f[2] = sqr(z[0]) + sqr(z[1]) - sqr(L);
}
string DAEsolverTest::getSuiteName(void)
{
    string suiteName = "DAEsolverSuite";
    return suiteName;
}
```

```
void DAEsolverTest::setUp(void)
{
    t0 = 0.0;
    tend = 100.0;
    n1 = 3;
    ptrSolver1 = new DAEsolver( n1, DAE_FCN(fcn1) );
}

void DAEsolverTest::tearDown(void)
{
    delete ptrSolver1;
}
}
```

5.2.3.3 Creating Testing Logic Files for the Test Suite

The test cases `testGetCVector` and `testSetHmaxThrow` are implemented in test logic files `daeSolverTest.cpp`.

The function `testGetCVector` aims to check if the function `getCVector` returns the expected "`c`" vector. It gets the "`c`" vector by calling the function `getCVector` of `DAEsolver` object, and then compare it with the expected result stored in "`t`" vector. If "c" vector is not equal to "t" vector, the macro `CPPUNIT_ASSERT` reports a failure to CppUnit.

The function `testSetHmaxThrow` intends to verify if the function `integrate`

throws `std::logic_error` exception when `hMax` is set less than `hMin`. It first calls the function `setHmax` with the parameter small than `hMin` to set the step-size less than the least stepsize, then it calls the `integrate` function. If the `integrate` function does not throw a `std::logic_error` exception, a failure are reported to CppUnit.

The snippet of `daeSolverTest.cpp` is:

```
#include <vector>
#include <algorithm>
#include <cppunit/config/SourcePrefix.h>
#include "daeSolverTest.h"
using namespace std;
using namespace daets;
namespace daetsTest{
/* DAEsolverTest::testGetCVector repro steps:
1) create a solver object (which has a non ill-posed
   sigmaMatrix)
2) input a correct vector C with n structure
3) check the result */
void DAEsolverTest::testGetCVector(void)
{
        vector<int> c(n1);
        ptrSolver1 -> getCVector(c);
        vector<int> t;
        t.push_back( 0 );
        t.push_back( 0 );
        t.push_back( 2 );
```

```
        CPPUNIT_ASSERT( equal(c.begin(), c.end(), t.begin())
            );
}
/* DAEsolverTest::testSetHmaxThrow repro steps:
1) create a solver object
2) call setHmax with parameter small than hMin
3) it throws std::logic_error exception */
void DAEsolverTest::testSetHmaxThrow()
{
        ptrSolver1 -> setHmax( ptrSolver1 -> getHmin() - (1e
            -15));
    SolverExitFlag flag;
        ptrSolver1 -> integrate( (*ptrSolution1), 100, flag);
}
}
```

5.2.3.4 Registering the Test Suite

Because this new created test suite only tests the single pubic interface `DAESolver` class, it belongs to `UnitTest` test suite group. The test suite need registering by `CPPUNIT_TEST_SUITE_NAMED_REGISTRATION` and `CPPUNIT_REGISTRY_ADD` as a member of `UnitTest` test suite group. `UnitTest` test suite group is then registered into the default test suite by the macro `CPPUNIT_REGISTRY_ADD_TO_DEFAULT`.

The snippet of `daeTest.h` is:

```
#ifndef DAETEST_H
#define DAETEST_H
#include "daeSolverTest.h"
CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(
        daetsTest::DAEsolverTest,
        daetsTest::DAEsolverTest::getSuiteName() );
CPPUNIT_REGISTRY_ADD(
        daetsTest::DAEsolverTest::getSuiteName(),
        "UnitTest" );
CPPUNIT_REGISTRY_ADD_TO_DEFAULT( "UnitTest" );
#endif
```

5.2.3.5 Selecting a Test Suite or a Test Suite Group to Execute

In the file `DAETest.cpp` of `DAETSTest` test project, the test suite or its test suite group can be selected to execute test cases.

This snippet of `DAETest.cpp` selects `DAEsolverTest` test suite to execute is:

```
#include <cppunit/TextOutputter.h>
#include <cppunit/TestResult.h>
#include <cppunit/TestResultCollector.h>
#include <cppunit/BriefTestProgressListener.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/TestRunner.h>
#include <iostream>
#include <string>
```

```
#include "daeTest.h"

using namespace std;
using namespace daetsTest;

int main(int argc, char** argv){
    CPPUNIT_NS::Test *suite =
        CPPUNIT_NS::TestFactoryRegistry::
            getRegistry(DAEsolverTest::getSuiteName())
                .makeTest();
```

This snippet selects UnitTest test suite group to execute

```
int main(int argc, char** argv){
    CPPUNIT_NS::Test *suite =
        CPPUNIT_NS::TestFactoryRegistry::
            getRegistry("UnitTest").makeTest();
```

This snippet selects all the test suites to execute

```
int main(int argc, char** argv){
    CPPUNIT_NS::Test *suite =
        CPPUNIT_NS::TestFactoryRegistry::
            getRegistry().makeTest();
```

5.2.3.6 Testing Result Output of CppUnit

A real example of CppUnit testing result output of DAETS testing project as follow.

```
!!!FAILURES!!! Test Results: Run: 163 Failures: 1 Errors: 0
1) test: N9daetsTest16CodeCoverageTestE::
   testModifiedPendulumForPow (F)
line: 651 codeCoverageTest.cpp
assertion failed - Expression: flag == daets::success
```

This information means ATF CppUnit executed 163 test cases, and 1 test case failed. The output also reports the name of the failed test case and the line of source code in the test case that led to this failure.

For the total output result of DAETS testing project, refer to Appendix B.

5.3 Code Coverage Analysis Tools

5.3.1 Introduction

If a tester wants to use black box testing to find all possible bugs, this method usually turns out to be infeasible. As a result, testers should further use white box testing method to supplement black box testing, so that they can find more defects in software.

White box testing requests testers to examine the internal business logic of a software instead of its public interface. Code Coverage Analysis of source code is one of the often used white box testing methods. Although this

method is not as strong as other complex white box methods e.g. branch coverage, it indeed works to supply supplementary test cases to find defects, which cannot be spotted by the black box testing.

5.3.2 What Code Coverage Analysis Can Achieve

Code coverage rate is an important way to evaluate the complete extent of test cases. Although it is not absolute, normally the higher the code coverage rate of software, the less defects may happen. For this reason, code coverage rate analysis can evaluate the quality of software to some extent.

Code coverage analysis also provides testers with clues about how to design new test cases to execute those uncovered statements. Although many test cases have already been designed using black box testing approach, it is usually impossible for those test cases to hit each statement in source code. In this case, code coverage analysis tools are introduced to help testers figure out the reason why test cases already designed cannot execute uncovered statements. Code coverage analysis tools can generate code coverage data, which show all the covered and uncovered statements of a program package in detail. After uncovered statements are located, some forward and backward analysis through the function calling chain of certain uncovered statements are carried out. The analysis results can often help testers discover the reasons why old test cases cannot cover these statements.

For instance, if a test case does not satisfy some conditions in a program, it can never hit the statements in that conditional branch. In most cases, it is the improper or incomplete input arguments of public interface functions that lead to these uncovered statements. To solve this problem, testers should trace the calling chain from the function, where these uncovered codes lie, until the public interface functions to figure out the reason why this condition cannot be satisfied by test cases. With this calling path analysis, testers are usually able to find the appropriate arguments for the public interface function, to trigger the located conditional branch. New test cases with arguments satisfying this condition can be added into test suites. As a consequence, code coverage rate also increases, duc to these new test cases, and testers are more confident of the quality of the software.

In addition, code coverage analysis can assist testers to locate software design and implementation defects in the source code. Sometimes the reasons resulting in uncovered code are software design defects or code implementation bugs. These software design defects and code implementation bugs may exist either just in uncovered statements or in somewhere of the calling chain of the uncovered code.

For example, some events or conditions can never be triggered due to design defects. In turn, those codes implemented to deal with the events can never be used or called. Testers sometimes may find that some business logic codes cannot be covered, even though the corresponding event or condition is

already triggered by a well-designed test case. This phenomenon is very likely due to bugs in the source code that make some condition become a tautology or never hold. In this case, testers should first trace the whole calling chain for uncovered statements. When testers make sure that the existed test cases ought to execute these uncovered statements with the tracing result, they should check, if it is bugs that give rise to the unexpected situation. Testers should be sensitive to those two situations mentioned above, and try to locate and get rid of this kind of defects and bugs by doing code coverage analysis.

Finally, code coverage analysis can help testers discover and remove dead code in source code. Dead code can also result in uncovered code. In the process of the software development, some out of date codes are often present. These out of date codes are called dead code. They are not the part of the calling chain of functions, and they can never be called or call other functions.

Several reasons can make dead code exist. For instance, when some developers update codes or functions of a business logic, they still hold the old version as the backup for the rolling back purpose. This operation often happens when a developer tries to fix a bug. After some time, even the developer himself may forget to delete the backup.

The other reason often occurs in large projects. In such projects, it is often a large team of programmers who simultaneously develop a system. Developers in this team are often requested to fix a bug located in some module owned

by others (maybe the owner of this module is absent or leaves this team). In this case, he would usually copy the original module and then slightly modify this copied module to try to fix the bug. Once new module can run correctly, he will stop without deleting the original module for the purpose of backup. However, more and more modules accumulate in the developing process of software. This finally gives rise to this situation: functions or blocks of codes related to the modules that have been modified cannot be triggered anymore. These functions or code blocks become dead codes in the end. Same things happen to the maintenance phase of software as well.

Using code coverage data and tracing the calling chain, testers can sometimes find certain functions or code blocks are not public interface functions, and they also never relate to other functions or code blocks. Testers can think this kind of codes as dead code. Based on the convention, testers must inform developers, owning this module, the dead code information. Only developers owning this module can make the final decision, if certain statements or functions are real dead code. In this way, testers can help developers mark or delete dead code, and both developers and testers can benefit from code coverage analysis.

5.3.3 How to Do Code Coverage

The test project for DAETS uses statement coverage analysis, which is the most common code coverage test method in the testing field. It is supported

by the build-in feature Gcov of GCC [8].

When test engineers want to do code coverage analysis with Gcov, they should compile the source file with the compiling flags `-fprofile-arcs` and `-ftestcoverage`. In the process of compiling with these two flags, GCC builds call graphs and tracks basic blocks for the testing target source file and then creates a new file to hold calling graphs and basic block information. This new created file owns the same main file name with its corresponding source file, but it also has a different extension name, .gcno.

Besides creating the gcno file, GCC also adds additional instructions into the binary codes of each basic block of source file. These instructions are counters that count how many times the corresponding basic block is executed.

After source code is compiled with the flags `-fprofile-arcs` and `-ftestcoverage`, the binary just compiled needs executing to achieve code coverage data. Files with the extension name .gcda are automatically created for each instrumented source file in the executing process. The gcda files own the same main file name with its corresponding source files too, like the gcno files. These gcda files hold code coverage information.

To do code coverage operation for the library DAETS, the following `CXXFLAGS` variable is defined in the makefile of DAETS project:

```
CXXFLAGS = -fprofile-arcs -ftest-coverage ...
```

After the source files of DAETS are compiled with code coverage option and

then libdaets.a file is built, the executable testing file DAETest is build and linked with libraries such as the instrumented DAETS package — `libdaets`, and code coverage library of Gcov — `libgcov`, by using the following `LDLIBS` variable for the makefile of CppUnitTest package:

```
LDLIBS = -lcppunit -ldaets -lgcov ...
```

After that the executable testing file `DAETest` is executed, the gcda files holding code coverage data are generated.

With these gcda files, testers can use Gcov [8] or Lcov [10] tools to deal with them and generate the code coverage report that human being can understand.

In the DAETS test project, Lcov is chosen to generate the code coverage report. Lcov is a graphical interface for the GCC Gcov. It extends Gcov with a set of Perl scripts. Lcov first uses Gcov to handle gcda files and then produces a friendly HTML output based on the textual analysis result of Gcov.

The following instructions are used in the makefile of DAETest test project that generate code coverage data:

```
lcov --directory ./ccsrc --capture --output-file DAETS.info
genhtml DAETS.info
```

The HTML output file displays as below:

Figure 5.5: Code Coverage Report

LTP GCOV extension - code coverage report

Current view:	directory		
Test:	DAETS.info		
Date:	2009-03-03	Instrumented lines:	2520
Code covered:	85.7 %	Executed lines:	2159

Directory name	Coverage		
/home/binqzhou/FADBAD++		33.3 %	21 / 63 lines
/home/binqzhou/cc/ccsrc		96.2 %	1882 / 1956 lines
/usr/include/c++/4.2		20.3 %	15 / 74 lines
bits		57.2 %	231 / 404 lines
ext		47.6 %	10 / 21 lines
i486-linux-gnu/bits		0.0 %	0 / 2 lines

Generated by: LTP GCOV extension version 1.5

5.3.4 Results of Code Coverage Analysis

Code coverage analysis for DAETS, which belongs to white box testing, effectively helped to supplement test cases designed by DDT, locate software defects deeply related to the business logic of SCS, and find dead code.

Code coverage analysis found some code in DAETS library, which are never hit by test cases designed by DDT.

One reason for this issue is that test cases designed by DDT lack the information of low level functions. In this case, we can either go through the calling chain from the public interface functions until the low level functions, or get advice from the developer to figure out how to design appropriate inputs to hit the related code. For example, one equation of the simple pendulum formula is $0 = f = x'' + x * \lambda$. The black box test case cannot trigger some source code in the function `Sigmamatrix` for we cannot figure out an appropriate input without reading source code. `Sigmamatrix` (§5.1 in [1]) is the $n \times n$ signature matrix $\Sigma = (\sigma_{ij})$ of a DAE system where:

n is the number of equations of the DAE system.

$$\sigma_{ij} = \begin{cases} order\ of\ the\ derivative\ to\ which\ the\ jth\ variable\ x_j \\ occur\ in\ the\ ith\ equation\ f_i;\ or \\ -\infty\ if\ x_j\ does\ not\ occur\ in\ f_i. \end{cases}$$

With the help of the developer of DAETS, we change the equation to
$0 = f = x'' + x * \lambda + \sin(x) - \sin(x) + \cos(x) - \cos(x)$.
New test cases implemented from this new equation can uncover the un-hit source code in sigmamatrix.

Source code defects prove to be the other reason that cannot trigger some code. In this case, we went through the whole calling chain paths from the piece of code never hit until the public interface functions. We should try to locate a condition that can never be satisfied. By fixing the defects in the

condition branch, we can find unexecuted code. In DAETS testing project, two code defects were located and corrected in the function `isInfRow` and `DAEsolution::printFixed()`.

Code coverage analysis also exposed dead code of the DAETS library. We discovered that some un-hit source code is not located in any calling chain beginning from public interface functions. These code can be thought of as potential dead code. After getting the verification from the developer, we deleted them. In this way, we removes 150 lines of dead code.

By executing code coverage analysis, new test cases are supplemented to verify more features of DAETS, software defects are found and corrected, and all the dead code is removed. In the end, we achieved code coverage rate from 78.9% (1515 instrumented lines are hitted by black-box test cases out of 1920 instrumented lines) to 96.1% (1845 instrumented lines are hitted by black-box test cases + white-box test cases out of 1920 instrumented line).

Chapter 6

Conclusion

From the descriptions of the preceding chapters, we can conclude that DDT, code coverage analysis, and software testing tool can indeed improve the quality of DAETS.

By executing static inspection, we verify the user guide and public interfaces of DAETS. In the inspection process, we found 7 issues related to document errors and software design defects etc. The quality of the user guide is improved by correcting these issue. By using DDT, we designed 126 black box test cases. These test cases helped us discover 18 software defects related to implementation errors of source code, design issues of public interface functions etc. They also supplemented the user guide with good practices and pitfalls information about the calling experience of public interface functions.

By executing code coverage analysis, we further designed 37 test cases to uncover the source code that black box test cases originally could not hit, found 2 software defects deeply related to the business logic of the source code, and deleted 150 line dead code. By using ATF, we automate the testing process. Automated testing improves software testing performance, and makes the testing result more accurate, consistent and trustable. It also make frequent regression testing possible, which can provide a quick feedback of the quality for each modification of source code.

Hence, domain experts grasping software testing techniques and software testing tools mentioned above can effectively improve the quality of SCS. However, domain experts should still remember one more thing: when domain experts perform the testing task for SCS, they should think from the user's perspective, not the expert's or developer's perspective.

Finally, we summarize our recommendation. The first work is unit testing. Based on current SE experience, the earlier defects are found, the lower the cost pays. Therefore, developers and testers must spend enough time to do unit testing. This work does not waste development time, but saves the time. SE experience proves unit testing can effectively improve the quality of software and the developer's confidence.

The second work is the testability problem of SCS. When domain experts design SCS or developer implement SCS, testers should cooperate with them to make the features or modules testable. An untestable module usually

means that it can cost more time and money to locate and fix potential defects. For example, testers should make sure that domain experts provide the testing interface for those private functions and properties when they begin to design a SCS.

Bibliography

[1] Nedialko S. Nedialkov, John D. Pryce. DAETS User Guide Version 1.0, 2008

[2] Paul Hamill. Unit Test Frameworks, O'REILLY, 2004

[3] Glenford J. Myers. The Art of Software Testing, Second ed., John Wiley & Sons, Hoboken, New Jersey, 2004

[4] Bo Einarsson, Ronald Boisvert, Françoise Chaitin-Chatelin, Ronald Cools, Craig Douglas, Kenneth Dritz, Wayne Enright, William Gropp, Sven Hammarling, Hans Petter Langtangen, Roldan Pozo, Siegfried Rump, Van Snyder, Elisabeth Traviesas-Cassan, Mladen Vouk, WilliamWalster, and Brian Wichmann. Accuracy and Reliability in Scientific Computing. SIAM, Philadelphia, PA, 2005.

[5] Gregory V. Wilson. Where's the real bottleneck in scientific computing? Scientists would do well to pick some tools widely used in the software industry. American Scientist, 94(1), 2006.

[6] Diane Kelly, Nancy Cote, Terry Shepard, "Software Engineers and Nuclear Engineers: Teaming up to do Testing", proceedings Canadian Nuclear Society Conference, St John New Brunswick, June 2007

[7] S. Smith, W. Yu. A Document Driven Methodology for Developing a High Quality Parallel Mesh Generation Toolbox, 2009

[8] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection, GNU Press, Boston, MA, 2003

[9] CppUnit Documentation. `http://cppunit.sourceforge.net/doc/lastest/index.html`

[10] LCOV. `http://ltp.sourceforge.net/coverage/lcov.php`

[11] Suely Oliveira and David E. Stewart. Writing Scientific Software: A Guide to Good Style. Cambridge University Press, New York, NY, USA, 2006.

[12] Rebecca Sanders, Diane Kelly. The Challenge of Testing Scientific Software.

[13] Term definition. `http://msdn.microsoft.com/en-us/library/aa292484(VS.71).aspx`

[14] 80-20 Rule. `http://en.wikipedia.org/wiki/Pareto_principle`

[15] Visio. `http://office.microsoft.com/en-us/visio/default.aspx`

Appendix A

Example of Test Cases Organization

When we used CppUnit to automate DAETS testing project, most reference materials only provided simple examples to show how to use CppUnit to automate testing processes. These examples are usually too simple to represent systematically how a real automated testing project is organized and implemented.

This appendix tries to provide a detailed example to show how a real automated testing project using CppUnit is organized and implemented.

The first two files `daeTest.h` and `daeTest.cpp` come from the DAETS testing project. The `daeTest.h` represents how to organize the hierarchical structure

of the test suites of the DAETS testing project. The file `daeTest.cpp` show how to select a test suite or a test suite group as the testing target, and how to execute the test target.

The following files `daeSolverTest.h` , `setDAESolverTestEnv.cpp` , and `daeSolverTest.cpp` in appendix are not the same as the real test files in the DAETS testing project. They only intend to show how to implement a test class. However, they represent enough details to implement a test class in CppUnit.

I hope that this appendix can provide helpful support for readers, who want to use CppUnit to automate their testing project.

A.1 daeTest.h

The whole `daeTest.h` file.

```
1  #define DAETEST_H
2  #include "daePointTest.h"
3  #include "daeSolutionTest.h"
4  #include "daeSolverTest.h"
5  #include "codeCoverageTest.h"
6  #include "torusIntegrationTest.h"
7  #include "chemakzoIntegrationTest.h"
8  #include "daeIntegBackForthTest.h"
9  #include "testDerivsIntegrationTest.h"
10 #include "pendulumSimpleIntegrationTest.h"
11 #include "laynewatsonIntegrationTest.h"
12 #include "vdpIntegrationTest.h"
```

```
13
14 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::DAEpointTest, daetsTest::
       DAEpointTest::getSuiteName());
15 CPPUNIT_REGISTRY_ADD(daetsTest::DAEpointTest::getSuiteName(), "UnitTest" );
16 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::DAEsolutionTest, daetsTest
       ::DAEsolutionTest::getSuiteName());
17 CPPUNIT_REGISTRY_ADD(daetsTest::DAEsolutionTest::getSuiteName(), "UnitTest")
       ;
18 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::DAEsolverTest, daetsTest::
       DAEsolverTest::getSuiteName());
19 CPPUNIT_REGISTRY_ADD(daetsTest::DAEsolverTest::getSuiteName(), "UnitTest");
20 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::CodeCoverageTest, daetsTest
       ::CodeCoverageTest::getSuiteName());
21 CPPUNIT_REGISTRY_ADD(daetsTest::CodeCoverageTest::getSuiteName(), "UnitTest"
       );
22
23 CPPUNIT_REGISTRY_ADD_TO_DEFAULT("UnitTest");
24
25 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::VdpIntegrationTest,
       daetsTest::VdpIntegrationTest::getSuiteName());
26 CPPUNIT_REGISTRY_ADD(daetsTest::VdpIntegrationTest::getSuiteName(), "
       IntegrationTest");
27 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::ChemakzoIntegrationTest,
       daetsTest::ChemakzoIntegrationTest::getSuiteName());
28 CPPUNIT_REGISTRY_ADD(daetsTest::ChemakzoIntegrationTest::getSuiteName(), "
       IntegrationTest");
29 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::TorusIntegrationTest,
       daetsTest::TorusIntegrationTest::getSuiteName());
30 CPPUNIT_REGISTRY_ADD(daetsTest::TorusIntegrationTest::getSuiteName(), "
       IntegrationTest");
31 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::DAEIntegBackForthTest,
       daetsTest::DAEIntegBackForthTest::getSuiteName());
32 CPPUNIT_REGISTRY_ADD(daetsTest::DAEIntegBackForthTest::getSuiteName(), "
       IntegrationTest");
33 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::TestDerivsIntegrationTest,
```

```
        daetsTest::TestDerivsIntegrationTest::getSuiteName());
34  CPPUNIT_REGISTRY_ADD(daetsTest::TestDerivsIntegrationTest::getSuiteName(), "
        IntegrationTest");
35  CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::
        PendulumSimpleIntegrationTest, daetsTest::PendulumSimpleIntegrationTest
        ::getSuiteName());
36  CPPUNIT_REGISTRY_ADD(daetsTest::PendulumSimpleIntegrationTest::getSuiteName
        (), "IntegrationTest");
37  CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(daetsTest::LaynewatsonIntegrationTest,
         daetsTest::LaynewatsonIntegrationTest::getSuiteName());
38  CPPUNIT_REGISTRY_ADD(daetsTest::LaynewatsonIntegrationTest::getSuiteName(),
        "IntegrationTest");
39
40  CPPUNIT_REGISTRY_ADD_TO_DEFAULT("IntegrationTest");
41  #endif
```

A.2 daeTest.cpp

The whole daeTest.cpp file.

```
1  #include <cppunit/TextOutputter.h>
2  #include <cppunit/TestResult.h>
3  #include <cppunit/TestResultCollector.h>
4  #include <cppunit/BriefTestProgressListener.h>
5  #include <cppunit/extensions/TestFactoryRegistry.h>
6  #include <cppunit/TestRunner.h>
7  #include <iostream>
8  #include <string>
9  #include "daeTest.h"
```

```
10
11 using namespace std;
12 using namespace daetsTest;
13
14 int main(int argc, char** argv)
15 {
16     //select all test suites to run
17     CPPUNIT_NS::Test *suite = CPPUNIT_NS::TestFactoryRegistry::getRegistry()
           .makeTest();
18
19     //select the single DAEsolverTest test suite to run
20     //CPPUNIT_NS::Test *suite = CPPUNIT_NS::TestFactoryRegistry::getRegistry
           (DAEsolverTest::getSuiteName()).makeTest();
21     //select the whole UnitTest test suite group to run
22     //CPPUNIT_NS::Test *suite = CPPUNIT_NS::TestFactoryRegistry::getRegistry
           ("UnitTest").makeTest();
23     //select the whole IntegrationTest test suite group to run
24     //CPPUNIT_NS::Test *suite = CPPUNIT_NS::TestFactoryRegistry::getRegistry
           ("IntegrationTest").makeTest();
25
26     CPPUNIT_NS::TestResult controller;
27     CPPUNIT_NS::TestResultCollector result;
28     controller.addListener(&result);
29     CPPUNIT_NS::BriefTestProgressListener progress;
30     controller.addListener(&progress);
31     CPPUNIT_NS::TestRunner runner;
32     runner.addTest( suite );
33     const std::string path = "";
34     runner.run(controller,path);
35     CPPUNIT_NS::TextOutputter outputter( &result, std::cout);
36     outputter.write();
37
38     //Return error code 1 if the one of test failed.
39     return result.wasSuccessful() ? 0 : 1;
40 }
```

A.3 daeSolverTest.h

The snippet of daeSolverTest.h.

```
1  #ifndef SOLVERTEST_H
2  #define SOLVERTEST_H
3  #include <cppunit/extensions/HelperMacros.h>
4  #include "DAEsolver.h"
5
6  namespace daetsTest{
7  class DAEsolverTest :public CPPUNIT_NS::TestFixture
8  {
9      CPPUNIT_TEST_SUITE( DAEsolverTest );
10     CPPUNIT_TEST( testGetCVector );
11     CPPUNIT_TEST_EXCEPTION( testSetHmaxThrow, std::logic_error );
12     CPPUNIT_TEST_SUITE_END();
13
14 public:
15     void setUp(void);
16     void tearDown(void);
17     static std::string getSuiteName(void);
18     void testGetCVector(void);
19     void testSetHmaxThrow(void);
20
21 private:
22     daets::DAEsolver *ptrSolver1;
23     double t0;
24     double tend;
25     int n1;
26 }; /*Class header file ends*/
```

```
27 }/*namespace ends*/
28 #endif
```

A.4 setDAESolverTestEnv.cpp

The snippet of setDAESolverTestEnv.cpp.

```
1  #include "daeSolverTest.h"
2  using namespace std;
3  using namespace daets;
4
5  namespace daetsTest{
6  /* The DAE functions below come from pendulumsimple.cc well-posed */
7  template <typename T>
8  static void fcn1(T t, const T *z, T *f, void *param) {
9      // z[0], z[1], z[2] are x, y, lambda.
10     const double G = 9.8, L = 10.0;
11     f[0] = Diff(z[0],2) + z[0]*z[2];
12     f[1] = Diff(z[1],2) + z[1]*z[2] - G;
13     f[2] = sqr(z[0]) + sqr(z[1]) - sqr(L);
14 }
15
16 string DAEsolverTest::getSuiteName(void)
17 {
18     string suiteName = "DAEsolverSuite";
19     return suiteName;
20 }
21
22 void DAEsolverTest::setUp(void)
```

```
23 {
24     t0 = 0.0;
25     tend = 100.0;
26     n1 = 3;
27     ptrSolver1 = new DAEsolver( n1, DAE_FCN(fcn1) );
28 }
29
30 void DAEsolverTest::tearDown(void)
31 {
32     delete ptrSolver1;
33 }
34 }/*name space ends*/
```

A.5 daeSolverTest.cpp

The snippet of daeSolverTest.cpp.

```
1  #include <vector>
2  #include <algorithm>
3  #include <cppunit/config/SourcePrefix.h>
4  #include "daeSolverTest.h"
5
6  using namespace std;
7  using namespace daets;
8
9  namespace daetsTest{
10 /* DAEsolverTest::testGetCVector repro steps:
11 1) create a solver object (which has a non ill-posed sigmaMatrix)
12 2) retrieve a vector C with n length from the solver object
```

```
13  3) setup t vector with the correct offset and then compare with C vector
14  4) check if t vector is the same with c vector */
15  void DAEsolverTest::testGetCVector(void)
16  {
17      vector<int> c(n1);
18      ptrSolver1 -> getCVector(c);
19      vector<int> t;
20      t.push_back( 0 );
21      t.push_back( 0 );
22      t.push_back( 2 );
23      CPPUNIT_ASSERT( equal( c.begin(), c.end(), t.begin() ) );
24  }
25
26  /* DAEsolverTest::testSetHmaxThrow repro steps:
27  1) create a solver object
28  2) call setHmax with parameter small than hMin
29  3) it should throws std::logic_error exception */
30  void DAEsolverTest::testSetHmaxThrow()
31  {
32      SolverExitFlag flag;
33      ptrSolver1 -> integrate( (*ptrSolution1), 100, flag);
34  }
35  }/*namespace ends*/
```

Appendix B

Result of DAETS Testing Project

A real example of CppUnit testing result output of DAETS testing project as follow.

```
N9daetsTest12DAEpointTestE::testCtrWithSolver : OK
N9daetsTest12DAEpointTestE::testCtrWithSolverThrow : OK
N9daetsTest12DAEpointTestE::testCtrWithDAESolution : OK
N9daetsTest12DAEpointTestE::testCtrWithDAEpoint : OK
N9daetsTest12DAEpointTestE::testSetX : OK
N9daetsTest12DAEpointTestE::testSetXOrderThrow : OK
N9daetsTest12DAEpointTestE::testSetXIndexThrow : OK
N9daetsTest12DAEpointTestE::testSetXUnPairThrow : OK
N9daetsTest12DAEpointTestE::testGetXOrderThrow : OK
N9daetsTest12DAEpointTestE::testGetXIndexThrow : OK
```

```
N9daetsTest12DAEpointTestE::testGetNumVariables : OK
N9daetsTest12DAEpointTestE::testGetNumDerivatives : OK
N9daetsTest12DAEpointTestE::testGetNumDerivativesThrow : OK
N9daetsTest12DAEpointTestE::testGetNumDerivativesThrow2 : OK
N9daetsTest12DAEpointTestE::testOpAssignWithParaDouble : OK
N9daetsTest12DAEpointTestE::testOpAssignWithDAEpointObject :
   OK
N9daetsTest12DAEpointTestE::
   testOpAssignWithDAEpointObjectChain : OK
N9daetsTest12DAEpointTestE::testOpAddAssign : OK
N9daetsTest12DAEpointTestE::testOpAddAssignChain : OK
N9daetsTest12DAEpointTestE::testOpAddAssignThrow : OK
N9daetsTest12DAEpointTestE::testOpSubAssign : OK
N9daetsTest12DAEpointTestE::testOpSubAssignThrow : OK
N9daetsTest12DAEpointTestE::testOpMultiAssign : OK
N9daetsTest12DAEpointTestE::testOpMultiAssignThrow : OK
N9daetsTest12DAEpointTestE::testOpDivAssign : OK
N9daetsTest12DAEpointTestE::testOpDivAssignThrow : OK
N9daetsTest12DAEpointTestE::testOpAdd : OK
N9daetsTest12DAEpointTestE::testOpAddThrow : OK
N9daetsTest12DAEpointTestE::testOpSub : OK
N9daetsTest12DAEpointTestE::testOpSubThrow : OK
N9daetsTest12DAEpointTestE::testOpMultiply : OK
N9daetsTest12DAEpointTestE::testOpMultiplyThrow : OK
N9daetsTest12DAEpointTestE::testOpDivide : OK
N9daetsTest12DAEpointTestE::testOpDivideThrow : OK
N9daetsTest12DAEpointTestE::testOpEqual : OK
```

```
N9daetsTest12DAEpointTestE::testOpEqualThrow : OK
N9daetsTest12DAEpointTestE::testOpNotEqual : OK
N9daetsTest12DAEpointTestE::testOpNotEqualThrow : OK
N9daetsTest12DAEpointTestE::testOpEqualandNotEqual : OK
N9daetsTest12DAEpointTestE::testNorm : OK
N9daetsTest15DAEsolutionTestE::testCtr : OK
N9daetsTest15DAEsolutionTestE::testCtrThrow : OK
N9daetsTest15DAEsolutionTestE::testSetGetT : OK
N9daetsTest15DAEsolutionTestE::testSetGetTNotThrow
******************************************************************************

This program contains IPOPT, a program for large-scale
   nonlinear optimization. IPOPT is released as open source
   under the Common Public License (CPL). For more
   information visit www.coin-or.org/Ipopt
******************************************************************************

: OK
N9daetsTest15DAEsolutionTestE::testSetTLogicThrow : OK
N9daetsTest15DAEsolutionTestE::testGetTLogicThrow : OK
N9daetsTest15DAEsolutionTestE::testGetTypeUnitialized : OK
N9daetsTest15DAEsolutionTestE::testGetTypeUnitialized2 : OK
N9daetsTest15DAEsolutionTestE::testGetTypeFree : OK
N9daetsTest15DAEsolutionTestE::testGetTypeFixed : OK
N9daetsTest15DAEsolutionTestE::testGetTypeIndexRangeThrow :
   OK
N9daetsTest15DAEsolutionTestE::testGetTypeIndexRangeThrow2 :
```

```
   OK
N9daetsTest15DAEsolutionTestE::testGetTypeOrderRangeThrow :
   OK
N9daetsTest15DAEsolutionTestE::testGetTypeOrderRangeThrow2 :
   OK
N9daetsTest15DAEsolutionTestE::testSetGetX : OK
N9daetsTest15DAEsolutionTestE::testSetXIndexRangeThrow : OK
N9daetsTest15DAEsolutionTestE::testSetXOrderRangeThrow : OK
N9daetsTest15DAEsolutionTestE::testSetXLogicThrow : OK
N9daetsTest15DAEsolutionTestE::
   testSetXAfterIntegrateIndexRangeThrow : OK
N9daetsTest15DAEsolutionTestE::
   testSetXAfterIntegrateOrderRangeThrow : OK
N9daetsTest15DAEsolutionTestE::testSetXPairRangeThrow : OK
N9daetsTest15DAEsolutionTestE::
   testSetXPairAfterIntegrateRangeThrow : OK
N9daetsTest15DAEsolutionTestE::testSetXRepeated : OK
N9daetsTest15DAEsolutionTestE::
   testSetGetXIndexRangeWithoutIntegrateThrow : OK
N9daetsTest15DAEsolutionTestE::
   testSetGetXIndexRangeWithIntegrateThrow : OK
N9daetsTest15DAEsolutionTestE::
   testSetGetXOrderRangeWithoutIntegrateThrow : OK
N9daetsTest15DAEsolutionTestE::
   testSetGetXOrderRangeWithIntegrateThrow : OK
N9daetsTest15DAEsolutionTestE::
   testSetGetXIndexRangeWithoutIntegrateThrow2 : OK
```

```
N9daetsTest15DAEsolutionTestE::
   testSetGetXOrderRangeWithoutIntegrateThrow2 : OK
N9daetsTest15DAEsolutionTestE::
   testGetXIndexRangeWithoutIntegrateThrow : OK
N9daetsTest15DAEsolutionTestE::
   testGetXOrderRangeWithoutIntegrateThrow : OK
N9daetsTest15DAEsolutionTestE::testGetXLogicThrow : OK
N9daetsTest15DAEsolutionTestE::
   testSetXUninitializedGetXLogicThrow : OK
N9daetsTest15DAEsolutionTestE::testUpdatePointWithDAEpoint :
   OK
N9daetsTest15DAEsolutionTestE::testUpdatePointWithDAEsolution
    : OK
N9daetsTest15DAEsolutionTestE::
   testUpdatePointAfterIntegrateNotThrow : OK
N9daetsTest15DAEsolutionTestE::
   testUpdatePointWithDAESolutionLogicThrow : OK
N9daetsTest15DAEsolutionTestE::
   testUpdatePointWithDAEpointLogicThrow : OK
N9daetsTest15DAEsolutionTestE::
   testUpdatePointDifferentShapeDAEsulotionLogicThrow : OK
N9daetsTest15DAEsolutionTestE::
   testUpdatePointDifferentShapeDAEpointLogicThrow : OK
N9daetsTest15DAEsolutionTestE::testSetFirstEntry : OK
N9daetsTest15DAEsolutionTestE::testSetOneStepMode : OK
N9daetsTest15DAEsolutionTestE::testSetOutputFunction : OK
N9daetsTest15DAEsolutionTestE::testGetCPUtime : OK
```

```
N9daetsTest15DAEsolutionTestE::testGetNumAccSteps : OK
N9daetsTest15DAEsolutionTestE::testGetNumRejSteps : OK
N9daetsTest15DAEsolutionTestE::testPrintSolutionInfo : OK
N9daetsTest13DAEsolverTestE::testGetSigmaMatrix : OK
N9daetsTest13DAEsolverTestE::
   testGetSigmaMatrixUninitializedThrow : OK
N9daetsTest13DAEsolverTestE::
   testGetSigmaMatrixWrongStrucThrow : OK
N9daetsTest13DAEsolverTestE::testGetSigmaMatrixWithZeroThrow
   : OK
N9daetsTest13DAEsolverTestE::
   testGetSigmaMatrixWithPositiveThrow : OK
N9daetsTest13DAEsolverTestE::testGetCVector : OK
N9daetsTest13DAEsolverTestE::testGetCVectorUninitializedThrow
    : OK
N9daetsTest13DAEsolverTestE::testGetCVectorWrongStrucThrow :
   OK
N9daetsTest13DAEsolverTestE::testGetDVector : OK
N9daetsTest13DAEsolverTestE::testGetDVectorUninitializedThrow
    : OK
N9daetsTest13DAEsolverTestE::testGetDVectorWrongStrucThrow :
   OK
N9daetsTest13DAEsolverTestE::testIsIllPosed : OK
N9daetsTest13DAEsolverTestE::testCtrWithIllPosedFcn : OK
N9daetsTest13DAEsolverTestE::testIsQuasilinear : OK
N9daetsTest13DAEsolverTestE::testGetStructuralIndex : OK
N9daetsTest13DAEsolverTestE::testGetNumDegsOfFreedom : OK
```

```
N9daetsTest13DAEsolverTestE::testPrintInfo : OK
N9daetsTest13DAEsolverTestE::testSetTol : OK
N9daetsTest13DAEsolverTestE::testSetTolOutLowerThrow : OK
N9daetsTest13DAEsolverTestE::testSetTolOutUpperThrow : OK
N9daetsTest13DAEsolverTestE::testGetErrorEstTypeDefault : OK
N9daetsTest13DAEsolverTestE::testGetErrorEstTypeAbs : OK
N9daetsTest13DAEsolverTestE::testGetErrorEstTypeRel : OK
N9daetsTest13DAEsolverTestE::testGetXTol : OK
N9daetsTest13DAEsolverTestE::testSetGetOrder : OK
N9daetsTest13DAEsolverTestE::testSetOrderUpperBoundThrow : OK
N9daetsTest13DAEsolverTestE::testSetOrderLowerBoundThrow : OK
N9daetsTest13DAEsolverTestE::testGetHmaxHmin : OK
N9daetsTest13DAEsolverTestE::testSetHmax : OK
N9daetsTest13DAEsolverTestE::testSetHmaxThrow Pay an
   attention:
ptrSolver1 -> getHmin() == 2.22045e-16 : OK
N9daetsTest13DAEsolverTestE::testSetHmaxAfterIntegrationThrow
Pay an attention: ptrSolver1 -> getHmin() == 2.27374e-13 : OK
N9daetsTest13DAEsolverTestE::
   testSetHmaxAfterIntegrationNotThrow : OK
N9daetsTest16CodeCoverageTestE::
   testCheckSizeDiffStrucLogicThrow : OK
N9daetsTest16CodeCoverageTestE::
   testIllposedDAESolverContructor : OK
N9daetsTest16CodeCoverageTestE::
   testDAESolverCheckInputUninitialzingT
*** Initialize: t : OK
```

```
N9daetsTest16CodeCoverageTestE::
   testDAESolverCheckInputUninitializedX
*** Initialize: variable derivative(s) x0 0 1 x1 0 1 : OK
N9daetsTest16CodeCoverageTestE::
   testDAESolverCheckInputWithtoofewdofX
*** Too few degrees of freedom at stage = -2
*** Fixed are: variable derivative x0 0 x1 0 : OK
N9daetsTest16CodeCoverageTestE::
   testPrintDAEinfoWithIllPosedDAEsolver : OK
N9daetsTest16CodeCoverageTestE::
   testprintDAEpointStructureWithIllPosedDAEsolver : OK
N9daetsTest16CodeCoverageTestE::
   testprintDAEtableauWithIllPosedDAEsolver : OK
N9daetsTest16CodeCoverageTestE::
   testPrintStatsWithoutIntegration : OK
N9daetsTest16CodeCoverageTestE::
   testPrintSolutionWithUninitalizedT : OK
N9daetsTest16CodeCoverageTestE::
   testPrintSolutionWithXUninitalized : OK
N9daetsTest16CodeCoverageTestE::
   testprintSolutionStateWithInitalizedX : OK
N9daetsTest16CodeCoverageTestE::
   testprintSolutionStateWithInitialConsistentX : OK
N9daetsTest16CodeCoverageTestE::testGetSigmaMatrixLogicThrow
   : OK
N9daetsTest16CodeCoverageTestE::
   testIntegSolverExitFlagSuccess : OK
```

```
N9daetsTest16CodeCoverageTestE::
   testIntegSolverExitFlagUninitialT
*** Initialize: t : OK
N9daetsTest16CodeCoverageTestE::
   testIntegSolverExitFlagUninitialXEXIT
at stage 0 exit flag = 9 : OK
N9daetsTest16CodeCoverageTestE::
   testIntegSolverExitFlagTooFewDOF
*** Too few degrees of freedom at stage = -2
*** Fixed are: variable derivative x0 0 x1 0 : OK
N9daetsTest16CodeCoverageTestE::
   testIntegSolverExitFlagDefault : OK
N9daetsTest16CodeCoverageTestE::testDAEpointSetXRangeThrow1 :
    OK
N9daetsTest16CodeCoverageTestE::testDAEpointSetXRangeThrow2 :
    OK
N9daetsTest16CodeCoverageTestE::
   testModifiedPendulumForAlphaSinCos : OK
N9daetsTest16CodeCoverageTestE::
   testModifiedPendulumForAlphaAssignmentOperator : OK
N9daetsTest16CodeCoverageTestE::
   testModifiedPendulumForAlphaUnaryAddOperator : OK
N9daetsTest16CodeCoverageTestE::
   testModifiedPendulumForAlphaDivideAssignmentOperator : OK
N9daetsTest16CodeCoverageTestE::
   testModifiedPendulumForSintesting
sin = 0 : OK
```

```
N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForCostesting
cos = 0 : OK
N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForTantesting
tan = 0 : OK
N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForSqrtesting
sqr = 0 : OK
N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForSqrttesting
sqrt = 0 : OK
N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForExptesting
exp = 0 : OK
N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForPowEXIT
at stage -2 exit flag = 11 : assertion
N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForLogtesting
log = 0 : OK N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForAsintesting asin = 0 : OK
N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForAcostesting
acos = 0 : OK
N9daetsTest16CodeCoverageTestE::
    testModifiedPendulumForAtantesting
```

```
atan = 0 : OK N9daetsTest16CodeCoverageTestE::
   testIllPosedProblemForcompOffsets : OK
N9daetsTest18VdpIntegrationTestE::
   vdpIntegrationTestIntegrating
VDP with mu=1.0e+00 tend = 50.0 t = 5.0000e+01 steps = 241 h
   = 2.89e-01 le = 5.99e-10 Integrating VDP with mu=1.0e+01
   tend = 20.0 t = 2.0000e+01 steps = 209 h = 1.74e-01 le =
   1.31e-09 Integrating VDP with mu=1.0e+02 tend = 20.0 t =
   2.0000e+01 steps = 2682 h = 8.15e-03 le = 8.73e-10 : OK
N9daetsTest23ChemakzoIntegrationTestE::
   chemakzoIntegrationTest
*** Significant correct digits: 8.5 : OK
N9daetsTest20TorusIntegrationTestE::torusIntegrationTest : OK
N9daetsTest21DAEIntegBackForthTestE::daeIntegBackForthTest :
   OK
N9daetsTest25TestDerivsIntegrationTestE::
   testDerivsIntegrationTest : OK
N9daetsTest29PendulumSimpleIntegrationTestE::
   pendulumSimpleIntegrationTest
Error in pendulumSimpleIntegrationTest.cpp 4.33431e-13 : OK
N9daetsTest26LaynewatsonIntegrationTestE::
   laynewatsonIntegrationTest
Error in laywnewatsonIntegrationTest.cpp 8.27072e-12 : OK

!!!FAILURES!!! Test Results: Run: 163 Failures: 1 Errors: 0
1) test: N9daetsTest16CodeCoverageTestE::
   testModifiedPendulumForPow (F)
```

```
line: 651 codeCoverageTest.cpp
assertion failed - Expression: flag == daets::success
```