A REGRESSION TESTING SYSTEM FOR GRID ADAPTIVE COMPUTATIONAL ENGINE

BY DHAVAL KAPADIA

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

New Brunswick, New Jersey

April, 2004

ABSTRACT OF THE THESIS

A Regression Testing System for Grid Adaptive Computational Engine

by Dhaval Kapadia

Thesis Director: Professor Manish Parashar

GrACE (Grid Adaptive Computational Engine) is an adaptive computational and data-management engine for enabling distributed adaptive mesh-refinement computations on structured grids. Constant updates to this software and its critical functionality presents a need for an efficient and effective method of testing it.
In this work, we have implemented regression testing of GrACE to ensure that changes made to it, such as adding new features or modifying existing features, do not adversely affect the current functionality of the software. We have studied and compared various regression testing systems and test case generation methodologies. A regression testing system has been developed using the DejaGnu test framework which uses Expect based scripting for the design of test cases. It is then analyzed for its effectiveness in testing GrACE. Various test cases have been identified for a representative subset of classes. A modification based approach is used to identify the part of software to be tested. Test case minimization and prioritization is implemented and evaluated for its effectiveness in testing.

This system of testing would give the developers confidence in adding new functionality to the software and also a systematic and scientific approach to the problem of effective software testing.

Acknowledgements

Dedication

To my Parents

Table of Contents

List of Figures

Chapter 1
Introduction

This chapter presents an overview of the thesis, it describes the problem statement and the contribution of the thesis and also presents the organization of the thesis.

1.1 Motivation

In this section, we address the motivation and advantages of this work.

1.1.1 What is software testing

Software testing is a challenging task [11] . It's purpose is to evaluate the software to demonstrate that it meets the requirements. Software is increasingly being used in our daily life as well as in safety critical systems where failure could be life threatening. This enforces the need for an improved systematic discipline of testing which must be scientifically justifiable.

1.1.2 Why is it important

The cause of errors [16] in software may be attributed to the following:

- The user executed untested code Because of time constraints, it's not uncommon for developers to release untested code – code in which users can stumble across bugs.
- The order in which statements were executed in actual use differed from that during testing. This order can determine whether software works or fails.

- The user applied a combination of untested input values. The possible input combinations that thousands of users can make across a given software interface are simply too numerous for testers to apply them all. Testers must make tough decisions about which inputs to test, and sometimes we make the wrong decisions.

- The user's operating environment was never tested. We might have known about the environment but had no time to test it. Perhaps we did not (or could not) replicate the user's combination of hardware, peripherals, operating system, and applications in our testing lab. For example, although companies that write networking software are unlikely to create a thousand-node network in their testing lab, users may create such networks.

- Incomplete or faulty specifications. If the initial specifications are different from the actual implementation, the final result will not be as expected. Testers must verify the requirements documents.

1.1.3 The Need for Regression Testing

Software inevitably changes however well conceived and well written it initially may be. Operational failures expose faults to be repaired. Mistaken and changed requirements cause the software to be reworked. New uses of old software yield new functionality not originally conceived in the requirements. The management of this change is critical to the continuing usefulness of the software. The new functionality added to a system may be accommodated by the standard software development processes. Regression [10] testing attempts to revalidate the old functionality inherited from the old version. The new version should behave exactly as the old except where new behavior is intended. Therefore, regression tests for a system may be viewed as partial operational requirements for new versions of the system. Figure 1 shows a typical example of a sequence of time intervals during the life of a software system.

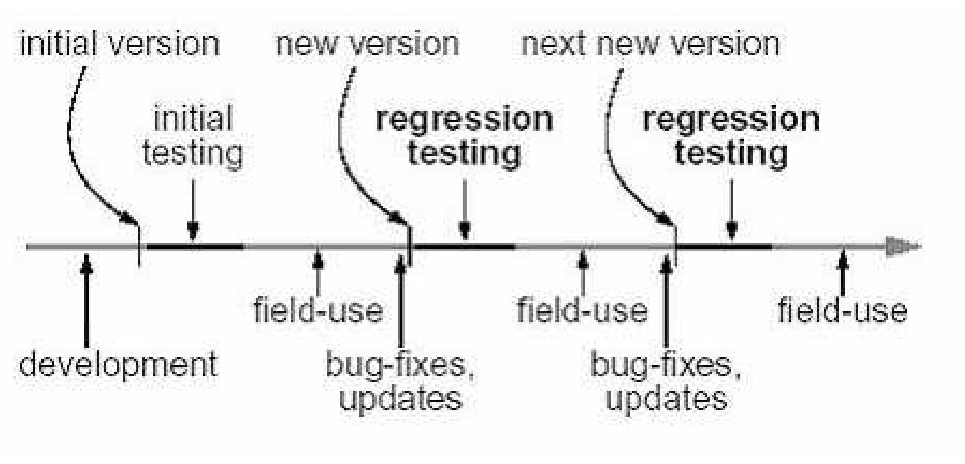

Figure 1.1: Regression Testing
[7]

1.2 Problem Statement

GrACE (Grid Adaptive Computational Engine) is an adaptive computational and data-management engine for enabling distributed adaptive mesh-refinement computations on structured grids. Constant updates to this software and its critical functionality presents a need for an efficient and effective method of testing it. Various test cases have to be identified and a test plan needs to be defined in order to efficiently manage the testing process. A testing system needs to be designed for achieving it.

1.3 Contributions of the thesis

- Various regression testing systems and test case generation methodologies are studied and compared.
- A regression testing system is developed using DejaGnu test framework and analyzed for its effectiveness in testing GrACE.
- Various test cases are identified for a representative subset of classes.
- Test case minimization and prioritization is implemented and evaluated for its effectiveness in testing.

1.4 Organization

The thesis is organized as follows: Chapter 1 gives the motivation of our work. It talks about the importance of software testing and the need for regression testing. It also gives an overview of building a regression testing system for GrACE. Chapter 2 talks about different types of regression testing mechanisms and gives a brief overview of the approach used by different projects in academic and commercial domain. Chapter 3 describes GrACE (Grid Adaptive Computation Engine) and some of its important classes. Chapter 4 presents Regression testing system. It also describes the tool DejaGnu / Expect. Chapter 5 describes architecture of the system and the method of choosing test cases using minimization and prioritization. It also describes the operation and implementation of the system. Chapter 6 presents the conclusions and comments on the future work.

Chapter 2
Background

Regression testing has received considerable research interest in recent years. In this chapter we summarize the recent efforts in this direction.

2.1 Regression Testing Systems

The software testing process, in general, consists of three phases. Unit testing exercises individual program units (e.g. procedures, classes) independently of called or calling routines. Integration testing exercises the interactions of program units. Unit and integration testing often require drivers or stubs to take the place of incomplete functionality. System testing exercises a complete software product or subsystem, typically within its normal operating environment. These familiar three phases are part of development testing, and are performed before a system is declared complete. After these three phases, a completed system enters 'maintenance mode'. In the maintenance stage of the software lifecycle, the software may continue to evolve. Once in use, new functionality may be desired or undetected faults may surface. This will require the completed, tested system to be modified. This in turn will require re-validation and re-testing [9]. This is regression testing. Regression testing serves several purposes. Its primary aims are to increase confidence in the correctness of a modified program, and locate errors in that program. It also helps preserve the quality and reliability of the existing system, and ensure its continued operation.

2.1.1 Regression Procedure

1. Select a subset of test cases to run on the modified program
2. Re-test the modified program and establish its correctness relative to the selected tests
3. If necessary, create new tests for the modified program
4. Update the existing test cases with the new tests from 3 and test history from 2.

2.1.2 Regression scope and test case selection

Most research on regression testing addresses one or both of two problems:

- how to select regression tests from an existing test suite (the regression test selection problem [6])
- how to determine the portions of a modified program that should be re-tested (the coverage identification problem).

There are three main philosophies to test selection in the literature:

- Minimization [17] approaches seek to satisfy structural coverage criteria by identifying a minimal set of tests that must be rerun to cover changed code.
- Coverage approaches are also based on coverage criteria, but do not require minimization. Instead, they seek to select all tests that exercise changed or affected program components.
- Safe methods attempt instead to select every test that will cause the modified program to produce different output than original program.

The difference in goals makes it difficult to compare and evaluate the test selection methods. Such evaluations are necessary, however, if we wish to choose techniques for practical application, or judge where additional research on selective re-test might be beneficial.

2.2 Related Research

2.2.1 Test Tube : A System for selective regression testing

TestTube [18] is a system that combines static and dynamic analysis to perform selective retesting of software systems written in C. TestTube first identifies which functions, types, variables and macros are covered by each test unit in a test suite. Each time the system under test is modified, TestTube identifies which entities were changed to create the new version. Using the coverage and change information, TestTube selects only those test units that cover the changed entities for testing the new version. TestTube has been applied to selective retesting of two software systems, an I/O library and a source code analyzer. Additionally TestTube has been adapted for selective retesting of nondeterministic systems where the main drawback is the unsuitability of dynamic analysis for identification of covered entities. TestTube gives an observed reduction of 50 percent or more in the number of test cases needed to test typical software changes.
TestTube methodology : The system is partitioned into basic code entities. They are defined such that they can be easily computed from the source code and monitored during execution. The execution of each test unit is monitored , its relationship is analyzed with the system under test and the subset of the code entities it covers is determined. When the system is changed, they identify the set of changed entities and then examine the previously computed set of covered entities for each test unit and check to see if any has changed. If none has changed, the test unit need not be rerun. If a test unit is rerun, its set of covered entities must be recomputed.

2.2.2 Incremental Regression Testing

In this paper [7], the authors propose some methods using which the test cases in the regression test suite whose outputs may be affected by the changes to the program may be identified automatically. Only these test cases need to be rerun during regression testing. The bulk of the cost of determining these tests is relegated to off-line processing. This problem of determining the test cases in a regression test suite on which the modified program may differ from the original program is the incremental regression testing problem.

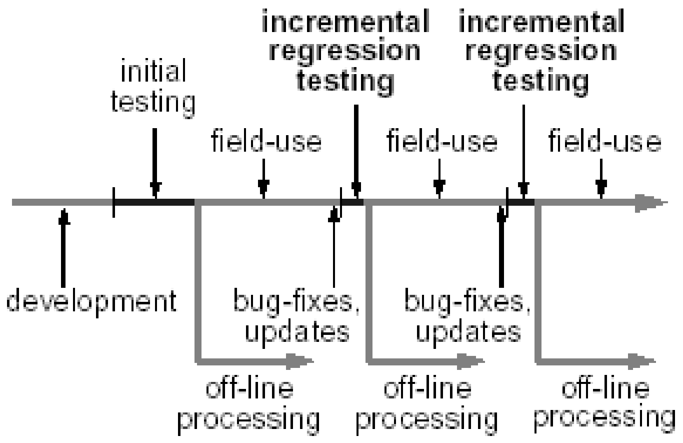

Figure 2.1: Incremental Regression Testing
[7]

The Execution slice technique: If, for a given test case, control never reaches that statement during the original program's execution, then it will never reach it during the new program's execution either. We refer to the set of statements executed under a test case as the execution slice of the program with respect to that test case. During the offline processing depicted in Figure, find the execution slices of the program with respect to all test cases in the regression test suite. Then, after the program is modified, rerun the new program on only those test cases whose execution slices contain a modified statement.

The Dynamic slice technique: A dynamic program slice is obtained by recursively traversing the data and control dependence edges in the dynamic dependence graph of the program for the given test case. During the offline processing depicted in Figure , the dynamic program slices with respect to the program output are found for all test cases in the regression test suite. Then, after the program is modified, the new program is rerun on only those test cases whose dynamic program slices contain a modified statement.

The Relevant slice technique: A relevant program slice is the set of statements that, if modified, may alter the program output for a given test case with respect to the program output for that test case. During the offline processing depicted in Figure, the relevant program slices with respect to the program output are found for all test cases in the regression test suite. Then , after the program is modified, the new program is rerun on only those test cases whose relevant slices contain a modified statement.

2.2.3 Prioritizing test cases for regression testing

Test case prioritization [15] techniques schedule test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal. In this paper, the authors discuss several techniques for using test execution information to prioritize test cases for regression testing. Various techniques for Prioritization for the rate of fault detection are as follows:

- Optimal : Prioritize to optimize the rate of fault detection
- Stmt-total : Prioritize in order of coverage of statements
- Stmt-addtl : Prioritize in order of coverage of statements not yet covered
- Branch-total : Prioritize in order of coverage of branches
- Branch-addtl : Prioritize in order of coverage of branches not yet covered

- FEP-total : Prioritize in order of total probability of exposing faults
- FEP-addtl : Prioritize in order of total probability of exposing faults, adjusted to consider effects of previous test cases

2.3 Summary

The Test Tube approach works well if the code entities are defined so that the partitioning of a software system can be done efficiently while still allowing effective reduction in the number of test cases that are selected. The expected maintenance actions on some kinds of systems actually favor a selective retesting strategy. In particular, when the maintenance actions are typically perfection and enhancement of specialized feature functions, the resulting modifications can require relatively small amounts of retesting. Test Tube employs a relatively coarse-grained analysis of the system under test, producing a reasonable and practical tradeoff between granularity of analysis and time/space complexity.

The incremental regression testing paper explores efficient methods of selecting small subsets of regression test sets. The amount of regression testing effort saved using the techniques discussed here depends on the nature of test cases in the regression test suite as well as the extent and the locations of the changes made. If the test cases are numerous and they each exercise small parts of the program's functionality then using these techniques should lead to greater savings. If, on the other hand, there are only a few test cases and each of them exercises most of the program's functionality then the methods will be less useful. The program locations where changes are made may also have a major effect on the amount of savings implied by using these techniques. A single change to an initialization statement that affects the program output for almost all test cases means almost all test cases must be rerun. On the other hand, even if changes are made to many parts of the program that are rarely executed by the regression tests, these techniques may mean significant savings.

In the test case prioritization research paper, the relative abilities of several techniques are empirically examined. The results suggested that these techniques can improve the rate of fault detection of test suites and that this result occurs even for the least sophisticated techniques. The result with respect to code-coverage-based techniques have immediate practical implications and that these can be leveraged for additional gains through prioritization.

We have utilized a combination of the above techniques for Regression testing of GrACE. After considering all the possible testing techniques, we have designed an effective testing system which leverages the features of the test methods and also gives a practically feasible solution considering the time and resources.

Chapter 3
Description of GrACE

3.1 Overview

GrACE (Grid Adaptive Computational Engine) [13] is an adaptive computational and data-management engine for enabling distributed adaptive mesh refinement [1] computations on structured grids. It builds on the Distributed Adaptive Grid Hierarchy (DAGH) infrastructure and extends it to provide a virtual, semantically specialized distributed (and dynamic) shared memory infrastructure with multifaceted objects specialized to distributed adaptive grid hierarchies and grid functions.

The Grid Adaptive Computational Engine consists of two components:

- A set of programming abstractions in which computations on dynamic hierarchical grid structures are directly implemental. The appropriate programming abstractions are a hierarchy of scalable distributed dynamic grids and a set of operations on this grid hierarchy. The operations include creation, partitioning, computations on the grid such as stencil operations, communication among grid partitions at a single level and communication among grids at different levels of the hierarchy.

- A set of distributed dynamic data-structures that support the implementation of the abstractions in parallel execution environments and preserve efficient execution while providing transparent distribution of the grid hierarchy across processing element execution environment.

3.2 Adaptive Mesh Refinement

Dynamically adaptive numerical techniques for solving differential equations provide a means for concentrating effort to computationally demanding regions. In the case of hierarchical AMR methods, this is achieved by tracking regions in the domain that require additional resolution and dynamically overlaying finer grids over these regions. Techniques based on AMR start with a base coarse grid with the lowest acceptable resolution that covers the entire computational domain. As the solution progresses, regions in the domain with high solution error and requiring additional resolution are flagged and finer grids are overlaid on the flagged regions of the coarse grid. Refinement proceeds recursively so that regions on the finer grid requiring higher resolution are similarly flagged, and even finer grids are overlaid on these regions. The resulting grid structure is a dynamic adaptive grid hierarchy. The figure below shows adaptive grid hierarchy for the classic Berger AMR formulation.

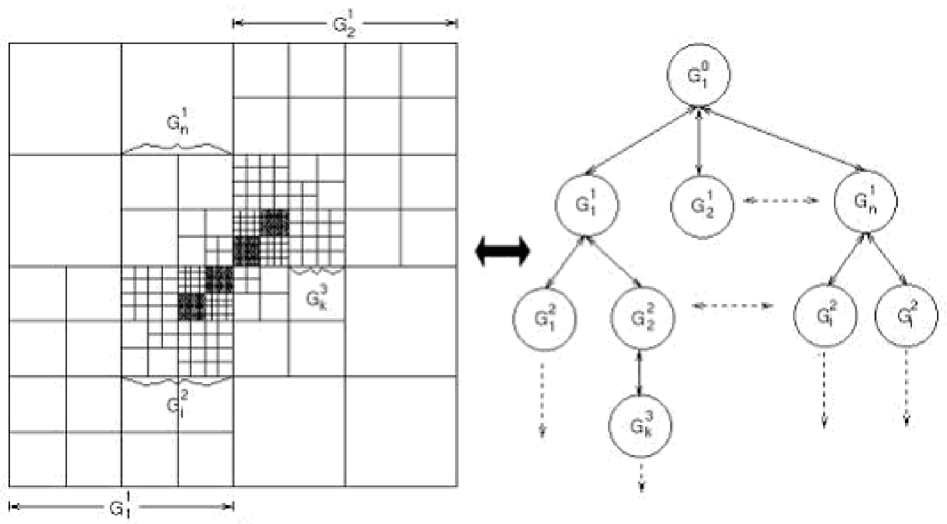

Figure 3.1: Adaptive Mesh Refinement
[13]

3.3 Programming Abstractions for Hierarchical AMR

For GrACE, three fundamental programming abstractions are developed, using the HDDA/DAGH data-structures, that can be used to express parallel adaptive computations based on adaptive mesh refinement (AMR) and multigrid techniques, as seen in the figure below. Our objectives are twofold: firstly, to provide application developers with a set of primitives that are intuitive for implementing the problem, i.e. application objects "-" abstract data-types, and secondly, a separation of data-management issues and implementations from application specific computations.

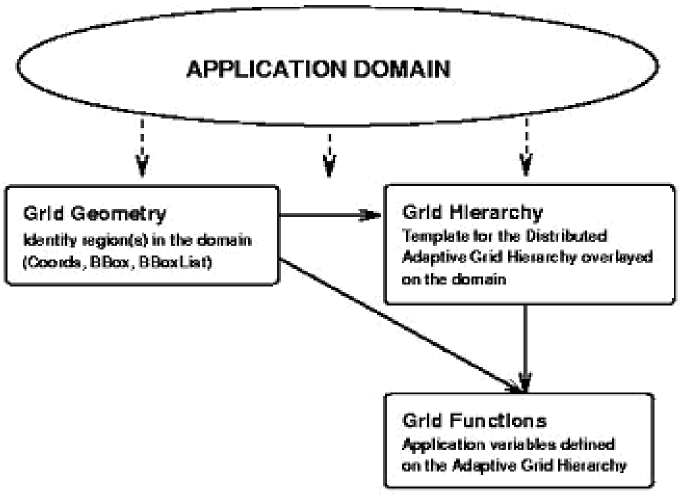
APPLICATION DOMAIN
Grid Geometry
Identify region(s) in the domain
(Coords, BBox, BBoxList)
Grid Hierarchy
Template for the Distributed
Adaptive Grid Hierarchy overlayed
on the domain
Grid Functions
Application variables defined
on the Adaptive Grid Hierarchy

Figure 3.2: Programming Abstractions for Hierarchical AMR
[13]

3.3.1 Grid Geometry Abstraction

The purpose of the grid geometry abstractions is to provide an intuitive means for identifying and addressing regions in the computational domain. These abstractions can be used to direct computations to a particular region in the domain, to mask regions that should not be included in a given operation, or to specify region that need more resolution or refinement. The grid geometry abstractions represent coordinates, bounding boxes and doubly linked lists of bounding boxes.

- Coordinates: The coordinate abstraction represents a point in the computational domain. Operations defined on this class include indexing and arithmetic/logical manipulations. These operations are independent of the dimensionality of the domain.

- Bounding Boxes: Bounding boxes represent regions in the computation domain and is comprised of a triplet: a pair of Coords defining the lower and upper bounds of the box and a step array that defines the granularity of the discretization in each dimension. In addition to regular indexing and arithmetic operations, scaling, translations, unions and intersections are also defined on bounding boxes. Bounding boxes are the primary means for specification of operations and storage of internal information (such as dependency and communication information) within DAGH.

- Bounding Boxes Lists: Lists of bounding boxes represent a collection of regions in the computational domain. Such a list is typically used to specify regions that need refinement during the regriding phase of an adaptive application. In addition to linked-list addition, deletion and stepping operation, reduction operations such as intersection and union are also defined on a BBoxList.

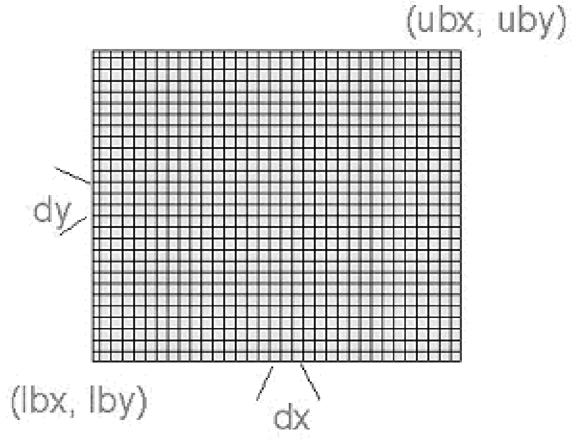

Figure 3.3: Grid Geometry
[13]

3.3.2 Grid Hierarchy Abstraction

The grid hierarchy abstraction represents the distributed dynamic adaptive grid hierarchy that underlie parallel adaptive applications based on adaptive mesh refinement. This abstraction enables a user to define, maintain and operate a grid hierarchy as a first-class object. Grid hierarchy attributes include the geometry specifications of the domain such as the structure of the base grid, its extents, boundary information, coordinate information, and refinement information such as information about the nature of refinement and the refinement factor to be used. When used in a parallel/distributed environment, the grid hierarchy is partitioned and distributed across the processors and serves as a template for all application variables or grid functions. The locality preserving composite distribution based on recursive space-filling curves is used to partition the dynamic grid hierarchy. Operations defined on the grid hierarchy include indexing of individual component grid in the hierarchy, refinement, coarsening, recomposition of the hierarchy after regriding, and querying of the structure of the hierarchy at any instant. During regriding, the repartitioning of the new grid structure, dynamic load-balancing, and the required data-movement to initialize newly created grids, are performed automatically and transparently.

3.3.3 Grid Function Abstraction

Grid Functions represent application variables defined on the grid hierarchy. Each grid function is associated with a grid hierarchy and uses the hierarchy as a template to define its structure and distribution. Attributes of a grid function include type information, and dependency information in terms of space and time stencil radii. In addition the user can assign special (Fortran) routines to a grid function to handle operations such as inter-grid transfers (prolongation and restriction), initialization, boundary updates, and input/output. These function are then called internally when operating on the distributed grid function. In addition to standard arithmetic and logical manipulations, a number of reduction operations such as Min/Max, Sum/Product, and Norms are also defined on grid functions. Grid Function objects can be locally operated on as regular Fortran 90/77 arrays.

3.4 Applications of GrACE

- H3expresso AMR : Numerical Relativity code that solves the full Einstein equations in 3D using a finite differencing conservative scheme and is parallel AMR operative
- Black-box Multigrid Solvers : Equations that routines will solve Lhuh = fh, "black-box" calling interface with multigrid flavors available
- Boson Star Application : Solves self-gravitating time-dependent Schrodinger equation in three spatial dimensions, with line-multigrid solvers used for update of Schrodinger field
- Texas Black Hole Evolution Code (TCODE):Black hole evolution code with leap frog time updates and second order methods at outer boundary replaced by first order methods
- 2D Laser-Plasma Interaction Fluid Code : Models the wave fields of a very short high-intensity laser pulse interacting with a plasma in a moving frame coordinate system, Lax Wendroff time updates, and 2D multigrid using a pointwise smoothing method

Chapter 4

The Regression Testing System

4.1 Architecture of the System

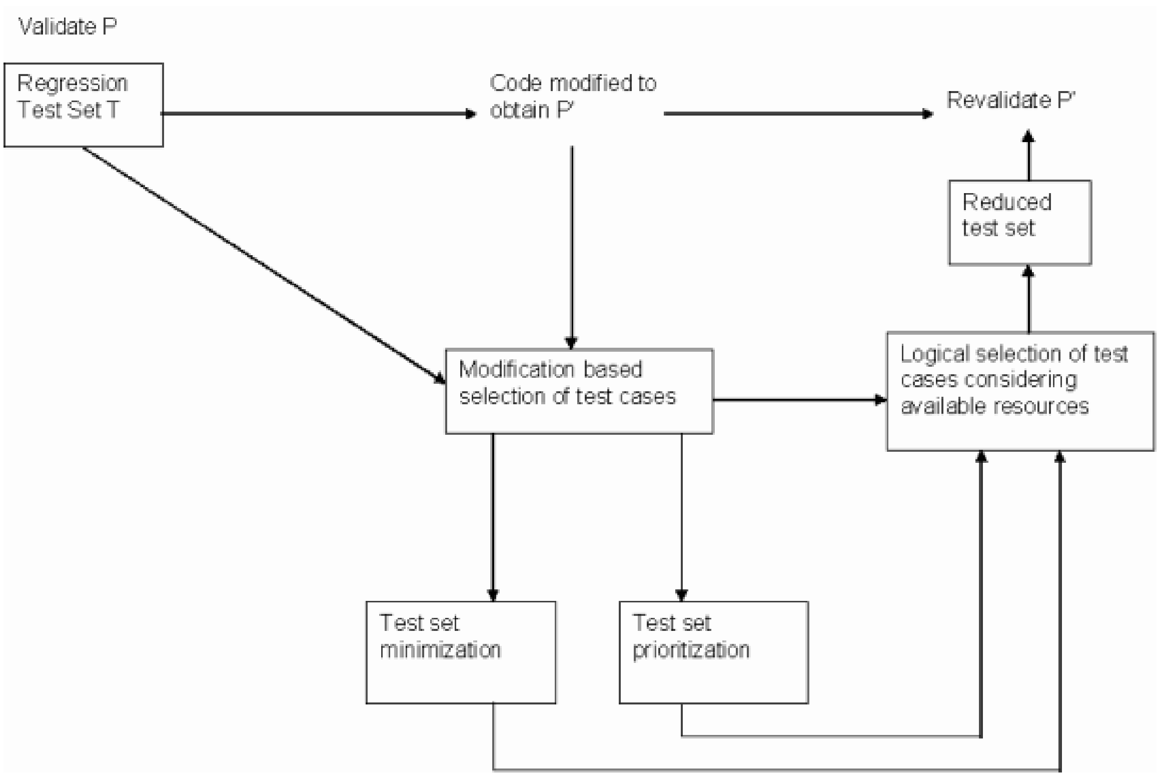
Validate P
Regression Test Set T
Code modified to obtain P'
Revalidate P'
Reduced test set
Modification based selection of test cases
Logical selection of test cases considering available resources
Test set minimization
Test set prioritization

Figure 4.1: Architecture of the system

The above diagram describes the basic architecture of the testing system followed by us. A modification [2] based technique is used for selection of code to be tested. A Regression test set is formed. The test cases are selected on the basis of test set minimization and prioritization. A reduced test set is obtained after considering the available resources.

4.2 DejaGnu

4.2.1 Introduction

DejaGnu is a collection of functions for Tcl and Expect for testing other programs and tools. Each program is tested by one or more test suites, which do the tests expressed as expect-scripts. Most times this tests can be done by sending the application a command and data and matching the result against the expected result. For the matching of the results the advanced regular expressions of Tcl can be used. DejaGnu is based on Expect, which itself is written in Tcl. All interaction with the application is done over Expect procedures, which use generic communication channels. So it's even possible to test applications on remote machines.

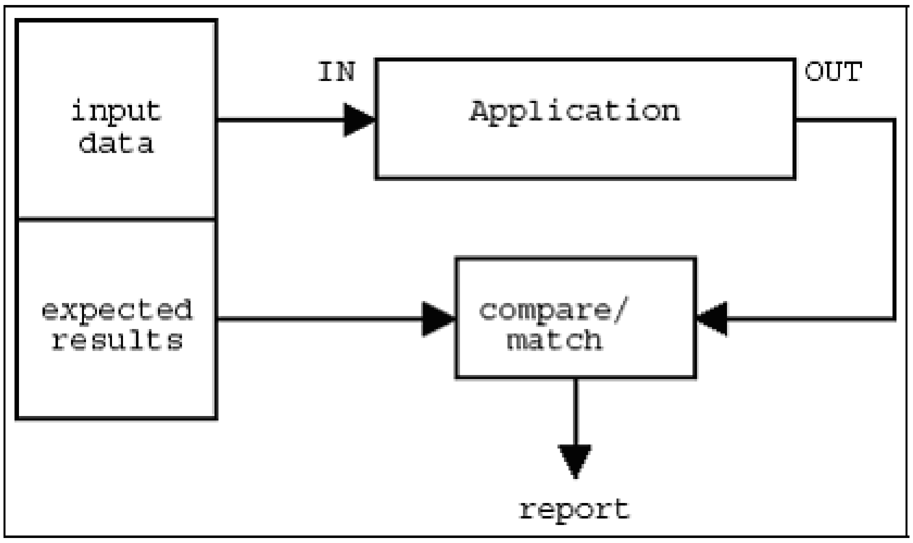

Figure 4.2: Principle of DejaGnu
[8]

4.2.2 Features

Since DejaGnu is based on Expect, it can be used to test batch oriented and also interactive programs, as long as they provide a stream based interface. These are mainly console based programs and tools but also libraries can be tested by using little wrappers. Here are some features: ” easy to write tests for batch oriented and for interactive programs ” provides layer of abstraction, necessary for cross-platform testing ” modular communication setup for remote testing ” unified, machine parseable but also human readable output format ” conforms to POSIX 1003.3 standard for regression test frameworks

4.2.3 Supplied Procedures

DejaGnu [8] [14] inherits a whole bunch of procedures from TCL and Expect and also provides a wide range of new functions for all forms of cross platform and netwide testing. Some of them are listed below.

- spawn : starts a program pass : declares a test to have passed
- fail : declares a test to have failed
- note : appends an informational message to the log file
- send : sends a string to the application
- expect : analyzes the output of the application
- getdirs : get a list of files and directories matching a pattern
- find : file search
- diff : finds differences of two or three files

There are some system and tool dependent procedures, which are used by DejaGnu if provided by the tester

- toolstart : starts the tool and initializes it. For a batch oriented tool it should be executed, leaving the output in the variable comp output

- toolload : loads something for a particular test case into the tool
- toolexit : cleaning up before DejaGnu exits
- toolversion : prints the version number for use in the summary report

4.2.4 Expect

Expect [12] [5] is a tool for automating interactive applications such as telnet, ftp, passwd, fsck, rlogin, tip, etc. Expect really makes this stuff trivial. Expect is also useful for testing these same applications. And by adding Tk, we can also wrap interactive applications in X11 GUIs. Expect can make easy all sorts of tasks that are prohibitively difficult with anything else. Expect is an absolutely invaluable tool - using it, we can automate various tasks quickly and easily.

4.2.5 Tcl

Tcl [4] stands for ”tool command language” and is pronounced ”tickle.” Tcl is actually two things: a language and a library. First, Tcl is a simple textual language, intended primarily for issuing commands to interactive programs such as text editors, debuggers, illustrators, and shells. It has a simple syntax and is also programmable, so Tcl users can write command procedures to provide more powerful commands than those in the built-in set. Second, Tcl is a library package that can be embedded in application programs. The Tcl library consists of a parser for the Tcl language, routines to implement the Tcl built-in commands, and procedures that allow each application to extend Tcl with additional commands specific to that application. The application program generates Tcl commands and passes them to the Tcl parser for execution. Commands may be generated by reading characters from an input source, or by associating command strings with elements of the application's user interface, such as menu entries, buttons, or keystrokes. When the Tcl library receives commands it parses them into component fields and executes built-in commands directly.

For commands implemented by the application, Tcl calls back to the application to execute the commands. In many cases commands will invoke recursive invocations of the Tcl interpreter by passing in additional strings to execute (procedures, looping commands, and conditional commands all work in this way).

Chapter 5
Implementation

We have focused on the testing with classes in the Grid Geometry package of the GrACE (Grid Adaptive computational engine).
Classes of the Grid Geometry package enable Bounding Box creation, modification and listing. The classes are as follows :

- Class Coords : A coordinate represents a point in the computational domain. Operations defined on this class include indexing and arithmetic/logical manipulations. These operations are independent of the dimensionality of the domain.
- Class BBox : Bounding boxes represent regions in the computation domain and is comprised of a pair of Coords defining the lower and upper bounds of the box and a step array that defines the granularity of the discretization in each dimension. In addition to regular indexing and arithmetic operations, scaling, translations, unions and intersections are also defined on bounding boxes.
- Class BBoxList : Lists of bounding boxes represent a collection of regions in the computational domain. Such a list is typically used to specify regions that need refinement during the regriding phase of an adaptive application. In addition to linked-list addition, deletion and stepping operation, reduction operations such as intersection and union are also defined on a BBoxList.

To conduct efficient and effective regression testing, we have followed the sequence given below:

1. A modification based test selection is used for deciding which part of the code is to be tested.
2. Minimization and Prioritization is used for selecting the test cases used for testing.

5.1 Testing Details

The details of choosing test cases has been described below. The minimization and prioritization for each test case has also been described.

The following two constructors were used for testing:

- public void BBox(void)
  Initializes a newly created BBox with constant limits for the size, step and dimensions of the Bounding Box.
- public void BBox(const int r, const int i, const int j, const int ii, const int jj, const int s)
  Initializes a newly created BBox of two dimensions between lower and upper bounds in each dimension of a constant step size from argument r, i, j, ii, jj, s. Parameters: r an const int - dimension of the Bounding Box i, ii are const int - lower bounds of Bounding Box in each dimension j, jj an const int - upper bounds of Bounding Box in each dimension s an const int - step size of the Bounding Box

The following functions have been tested with the standard input range for the dimensions of BBox :

- public BBox growupper(BBox const &bbox, const int c)
  Returns a new BBox object by increasing only the upper bounds by a constant

amount having the same step size in all the dimensions. Parameters: bbox an BBox const& - argument bbox to be modified. can const int - value used for modification.

- public BBox growlower(BBox const &bbox, const int c)

  Returns a new BBox object by increasing only the lower bounds by a constant amount having the same step size in all the dimensions. Parameters: bbox an BBox const& - argument bbox to be modified. c an const int - value used for modification.

The increase along different dimensions is chosen depending on the bounding values along each dimension :

- public BBox growupperbydim(BBox const &bbox, const short* c)

  Returns a new BBox object by increasing the upper bounds of BBox by different amounts in different dimensions. But may not have the same step size in all dimensions. Parameters: bbox an BBox const &- argument bbox to be modified. c an const short* - array of values, to modify in each dimension.

- public BBox growlowerbydim(BBox const &bbox, const short* c)

  Returns a new BBox object by increasing the lower bounds of BBox by different amounts in different dimensions. But may not have the same step size in all dimensions. Parameters: bbox an BBox const & - argument bbox to be modified. c an const short* - array of values, to modify in each dimension.

- public BBox shrinkupperbydim(BBox const &bbox, const short* c)

  Returns a new BBox object by decreasing the upper bounds of BBox by different amounts in different dimensions. But may not have the same step size in all dimensions. Parameters: bbox an BBox const - argument bbox to be modified. c an const short* - array of values, to modify in each dimension.

- public BBox shrinklowerbydim(BBox const &bbox, const short* c)
  Returns a new BBox object by decreasing the lower bounds of BBox by different amounts in different dimensions. But may not have the same step size in all dimensions. Parameters: bbox an BBox const& - argument bbox to be modified. c an const short* - array of values, to modify in each dimension.

- public BBox growbydim(BBox const &bbox, const short* c)
  Returns a new BBox object by increasing the lower and upper bounds of BBox by different amounts in different dimensions. But may not have the same step size in all dimensions. Parameters: bbox an BBox const&- argument bbox to be modified. c an const short* - array of values, to modify in each dimension.

- public BBox shrinkbydim(BBox const &bbox, const short* c)
  Returns a new BBox object by decreasing the lower and upper bounds of BBox by different amounts in different dimensions. But may not have the same step size in all dimensions. Parameters: bbox an BBox const& - argument bbox to be modified. c an const short* - array of values, to modify in each dimension.

The lower and upper bounds are used considering the frequently used values for the bounds. Two functions with different parameters are tested for preciseness :

- public BBox accrete(BBox const &bbox, const int c)
  Returns a new BBox object by increasing the lower and upper bounds by a constant amount having the same step size in all dimensions. Parameters: bbox and BBox const& - argument bbox to be modified. c an const int - value used for modification.

- public BBox accrete(BBox const &bbox, Coords const &c)
  Returns a new BBox object by increasing the lower and upper bounds by a constant amount having the same step size in all dimensions. Parameters:

bbox an BBox const& - argument bbox to be modified. c an Coords const - coordinates used for modification.

The following two different parameters are used to set the lower and upper values of the co-ordinates:

- bb.setlower(crds)
- bb.setlower(temp)
- bb.setupper(crds)
- bb.setupper(temp)
- bb.setempty()
- bbox.empty() is used to check whether the bbox is empty.

The following different parameters are used to set the co-ordinates of bbox :

- bb.setbbox(temp,temp)
- bb.setbbox(crds,crds)
- bb.setbbox(tempbox)

The functions which return size and bottom are tested as follows :

- bb.size()
- bb.bottom()

The operators have been tested with previous values of bbox co-ordinates and they return the added or multiplied values :

- bbTemp = bb1 + bb2
- bb1+=bb2
- bbTemp=bb1*bb2
- bb1*=bb2

The following function has been tested using standard values of dimensions of the bounding box :

- public int inside(const int i, const int j, const int k) const Determines the i, j and k dimension of a point to be inside the Bounding Box. Returns 1 if the coordinate obtained is contained by the Bounding Box, could on the edge. Parameters: i, j and k an const int - are the position in each dimension.

The following function has been tested using the previously defined values of the bounding box:

- public int mergable(BBox const &rhs, const int d, const int olap) const Checks the Bounding Box for compatibility and determines whether the Bounding Box is mergable for all the three dimension conditionality. Parameters: rhs an BBox const - Bounding Box to be checked for mergability d an const int - limits of the Bounding Box olap an const int - overlap

The following function has been tested using the frequently used values for increasing the bounding box:

- public void coarsen(const int by) Increase the step size in all dimensions of BBox by a constant amount. Parameters: by an const int - amount to be magnify.

5.2 Test Cases

The following is the expect script which has been used to test the functions of bbox class. The inputs to the functions have been determined considering bounding cases.

The outputs given by the functions are compared with the expected outputs and errors are reported. The expect scripts are managed by the dejagnu test framework. Required test scripts can be processed in any order on any platform by the test framework. The order for testing individual functions is determined considering the priority of each function.

set testdata

"Test1 growupper growlower growupperbydim growlowerbydim shrinkupperbydim shrinklowerbydim growbydim shrinkbydim accrete1 accrete2 setlower setupper setempty setbbox setbbox setbbox" "50 2 4 1 2 1 2" " ^ 70 60 22 24 12 14 18 16 8 6 8 24 9 22 12 16 11 18 8 22 8 22 5 25 5 25 50 50 50 50 5 8 5 8 1000000000 -1000000000 5 8 5 8 50 50 50 50 2 10 10 20 20 121 -120 2 5 5 20 20 2 5 5 20 20.*$"

"Test2 sameasabove" "60 1 2 4 1 1 2" " ^ 80 70 21 22 11 12 19 18 9 8 9 22 6 21 11 18 14 19 8 22 8 22 5 25 5 25 50 50 50 50 5 8 5 8 1000000000 -1000000000 5 8 5 8 50 50 50 50 2 10 10 20 20 121 -120 2 5 5 20 20 2 5 5 20 20.*$"

global TESTBBOX foreach pattern $testdata eval "spawn $TESTBBOX [lindex $pattern 1]" expect -re [lindex $pattern 2] pass [lindex $pattern 0] default fail [lindex $pattern 0]

Similarly, The Coords, BBoxList and DCoords classes are processed.

Chapter 6
Conclusions and Future Work

6.1 Conclusion

Effective regression testing is a trade-off between the number of regression tests needed and the cost. The greater the number of regression tests, the more complete the program revalidation. However, this also requires a larger budget and greater resources which may not be affordable in practice. Running fewer regression tests may be less expensive, but has the potential of not being able to ensure that all the inherited features still behave as expected, except where changes are anticipated.

In the system implemented by us, we have tried to balance the time and effort to maximize the effectiveness of regression testing. An optimized methodology has been developed comprising of modification, minimization and prioritization of test cases and a testing framework has been utilized. This would give the developers confidence in adding new code if the testing system is utilized.

6.2 Future Work

We have studied the feasibility of utilizing an expert systems tool for automating and selecting relevant test cases depending on the system sensitive values. CLIPS [3] expert systems software has been identified as a good fit for integrating this functionality. Implementing this can reduce the manual selection of test cases considerably and can provide a faster and more precise solution to the testing problem.

References

[1] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic partial Differential Equations. *Journal of Computational Physics*, 53:484–512, 1984.

[2] B.Sherlund and B.Korel. Modification oriented regression testing. In *Proceedings of the Fourth International Software Quality Week*, pages 149–158, San Francisco, CA, 1991.

[3] Clips: An expert systems tool. . http://www.ghg.net/clips/CLIPS.html.

[4] Developer exchange. Tool command language. . www.tcl.tk.

[5] David L. Fisher. Advanced programming in expect : A bulletproof interface, Nov 1999.

[6] G.Rothermel. *Efficient, Effective Regression Testing using Safe Test Selection Techniques*. PhD thesis, Department of Computer Science, Clemson University, Clemson, SC, May 1996.

[7] E.W. Krauser H.Agarwal, J.R. Horgan and S.A. London. Incremental regression testing. In *International Conference on Software Maintenance*, pages 348–357. IEEE, 1993.

[8] Free Software Foundation Inc. A framework for testing other programs. . http://gnu.org/software/dejagnu.

[9] F.Raji K.F. Fischer and A.Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference*. IEEE Computer Society Press, 2000.

[10] H.K.N. Leung and L.White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, October 1989.

[11] Glenford J. Myers. *The Art of Software Testing*. Wiley Interscience, 1979.

[12] National Institute of Standards and Technology. A scripting tool. . http://expect.nist.gov.

[13] M. Parashar. GrACE - A Grid Adaptive Computation Engine. . http://www.caip.rutgers.edu/TASSL/Projects/GrACE.

[14] Randolf Rotta. Dejagnu - a short introduction. , July 2002. http://home.landau-gym.de/ randolf/linux/dejagnu/dejagnu.pdf.

[15] A. Malishevsky S. Elbaum and G. Rothermel. Prioritizing test cases for regression testing. In *International Symposium for Software Testing and Analysis*, pages 102–112, August 2000.

[16] IEEE Computer Society. IEEE 1044 - standard classification for software errors, faults and failures. *IEEE Computer Society*, 1994.

[17] S.London W.E.Wong, J.R.Horgan and A.P.Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th IEEE International Conference on Software Engineering*, pages 41–50, Seattle, WA, April 1995.

[18] D. Rosenblum Y.F. Chen and K.P. Vo. Testtube : A system for selective regression testing. In *Proc. 16th International Conference on Software Engineering*, pages 211–220, May 1994.