Test Plan (TP)
Rate Adjustment by Managing Inflows (RAMI)
Team M

Daniel Connor (connor) Chris Edwards (caedwa) Rod Howard (rihoward)
Maya Muthuswamy (mayadm) Lars Yencken (lljy)

October 30, 2002

Maintained By: Daniel Connor (connor@students.cs.mu.oz.au)
Version: 1.1

Abstract

A testing plan for the development of RAMI, a TCP/IP flow control module for the Linux kernel and a suite of network evaluation utilities.

Contents

1 Introduction 4
- 1.1 Purpose 4
- 1.2 Scope 4
- 1.3 Intended Audience 4
- 1.4 Personnel 4
  - 1.4.1 Development Team 4
  - 1.4.2 Client 4
  - 1.4.3 Supervisor 5
- 1.5 Overview of testing strategy 5
- 1.6 Overall Test Report 6
- 1.7 Automated Testing Script for SAM 6
- 1.8 Critical Units 6
  - 1.8.1 SAM Critical Units 6
  - 1.8.2 FCM Critical Units 6

2 Bugs 7
- 2.1 Defect Severity 7
- 2.2 Defect Reporting 7
- 2.3 Corrective Action 7

3 Informal Testing 8

4 Unit Testing 9
- 4.1 General Strategy 9
- 4.2 Black Box Testing 9
  - 4.2.1 Equivalence Partitioning 9
  - 4.2.2 Boundary Value Analysis 10
  - 4.2.3 Random Selection 10
- 4.3 Glass Box Testing 10
- 4.4 Naming Conventions for Unit Tests 10
- 4.5 Procedure for Unit Testing 10
- 4.6 Unit Testing Reports 11

5 Integration Testing 13
- 5.1 General Strategy 13
- 5.2 Naming Conventions for Integration Tests 13
  - 5.2.1 Breakdown of SAM 13
  - 5.2.2 Thread Based Integration 13
  - 5.2.3 Bottom Up Integration 14
  - 5.2.4 Top Down Integration 14
- 5.3 Order of Integration 15
- 5.4 Procedure for Integration Testing 16
- 5.5 Test Case Selection 17
- 5.6 Integration Test Reports 17
- 5.7 Regression Testing 17

6 Testing Procedure For Logger 18

7 GUI Testing 19
- 7.1 GUI Test Reports 19

8 Function Testing 20
- 8.1 Function Test Reports 20

9 System Testing **21**
9.1 General Strategy 21
9.2 System Test Reports 21

10 Installation Testing **22**
10.1 Installation Test Reports 22

11 Documentation Testing **23**

12 Acceptance Testing **24**

13 Testing procedures for FCM **25**
13.1 Stress Testing 25
13.1.1 Precheckin Testing 25
13.1.2 Nightly Testing 25
13.1.3 Standard Test Load 25
13.2 Correctness Testing 25
13.3 System Testing 26
13.4 Installation Testing 26
13.5 Acceptance Testing 26

14 Appendices **27**
14.1 PyUnit 27

List of Figures

1 Shows the modified MVC architecture used for SAM, including the decomposition of the model module into Model and Analysis modules. 14

1 Introduction

1.1 Purpose

The purpose of this document is to outline the testing procedures and methods that will be used to verify and validate RAMI with respect to the requirements described in the Software Requirements Specification (SRS), with respect to the interfaces as described in the Software Architecture Design Document (SADD) and also with respect to RAMI's behavior, outlined in the use cases in the Detailed Design Document (DDD).

1.2 Scope

The project RAMI allows the optimization and monitoring of low-bandwidth links, including graphing and analysis of data logged. Its functionality is split into two main components, Flow Control Module (FCM), which handles optimization of the links, and Statistical Analysis Module (SAM), which handles monitoring and data analysis.

Since the design of FCM forms part of its requirements, the breakdown into modules has, in effect, already been done. FCM will consist of two Linux kernel modules: one at the Receiver, and one at the Router, and an additional module at the Router for logging of data.

The design of SAM is object-oriented, and is based around the Model-View-Controller (MVC) design pattern, with an additional class called Logger that is responsible for the logging of data. More details about the breakdown of SAM and the order in which it will be built (and integrated) can be found in section 5.

The techniques, tools and procedures outlined in this document will be used to test the correctness of FCM and SAM with respect to the requirements.

Due to the difference between FCM and SAM, different testing techniques will be used for each. Section 13 will describe the approach to testing FCM. All other sections, unless otherwise stated, apply only to SAM.

1.3 Intended Audience

This document is written for the developers of RAMI (see section 1.4.1) and also any end maintainers of RAMI, including the Client.

1.4 Personnel

This section gives the contact details for all the people involved in the development of RAMI.

1.4.1 Development Team

The developers of RAMI are Team M, consisting of:

Name	Username	Home Ph	Mobile Ph
Maya Muthuswamy	mayadm	9890 3608	0409557919
Daniel Connor	connor	9435 9223	0402143812
Chris Edwards	caedwa	9830 5152	0409411438
Lars Yencken	lljy	9650 7794	0404028880
Rod Howard	rihoward	9853 5853	0409543008

1.4.2 Client

The Client for RAMI is:

Lachlan Andrew
Dept of Electrical Engineering
The University of Melbourne

Phone: 8344 3816
email: lha@unimelb.edu.au

1.4.3 Supervisor

The supervisor for RAMI is:

Mark Ng
Dept of Computer Science & Software Engineering
University of Melbourne

Phone: 8344 9140
email: markn@cs.mu.oz.au

1.5 Overview of testing strategy

Testing begins at the lowest level and moves on to higher levels as the project progresses. This is the overview of the testing strategy that will be used:

- The code for SAM will be written in Python, which is an Object Oriented language. Therefore, the basic unit of code is a class. As coding of each class progresses, the author is expected to carry out Informal Testing (see section 3). The aim of Informal Testing is to ensure the code fulfills its methods defined so far, and for the author to be reasonably confident in the code's functionality before running Unit Tests (see next dot point). Also, it ensures that major design flaws are spotted and fixed early. Since this form of testing is informal, and procedures for it may vary between individuals, there is no set procedure or documentation for Informal Testing.
- Prior to any formal testing, static analysis must be performed on the code. This will be done via the use of `pychecker`, which is a program that finds common bugs and mistakes in Python code, similar to the errors found in, for example, a C compiler. `pychecker` is run via the testing script for SAM (see section 1.7).
- The lowest level of formal testing is Unit Testing (see section 4). This is performed once the coding of a feature of a class is complete, and Informal Testing has been carried out by the author. Unit Testing is designed to ensure that each individual class of RAMI performs its function(s) correctly as a discrete, independent unit.
- Unit tested classes are then combined together incrementally to form the complete system. This is the next level of testing - Integration Testing (see section 5). The purpose of Integration Testing is to ensure that classes interface correctly with each other.
- The next level of testing is System Testing (see section 9). System Testing tests the final product as a whole for defects. This step consists of GUI Testing (see section 7) and Documentation Testing (see section 11).
- Function Testing (see section 8) is the next level of testing and ensures that the product complies with the requirements set out in the SRS.
- Installation Testing (see section 10) will be performed once the product is fully integrated. This step will ensure that the product can be installed onto the platform it is designed for.
- The highest level of testing is Acceptance Testing (see section 12). This is to consist of the core part of the Function Testing step, along with a similar procedure for non-functional parts of the SRS. It is to be performed with the Client to demonstrate that the product satisfies the requirements in the SRS and use cases in the DDD.

1.6 Overall Test Report

Test cases are to be stored under CVS in the directory `src/test`, so that tests will be checked out along with source code.
An overall test report will be stored in CVS under `src/test`. This report will be named `testprog.txt`. The test report will list all the different unit, integration and system tests and their status (such as: Not yet written, Not yet run, Not yet passed, Passed). When a new test is written, it is the responsibility of the author of the test to update the test report. When the status of a test changes (such as when a piece of code passes a test case it previously did not pass), it is the responsibility of the author of the code that has changed to update the test report file.

1.7 Automated Testing Script for SAM

To ease performing static analysis (through the use of the tool `pychecker`) and unit and integration tests on classes in SAM, a shell script will be written. This script, entitled `runtests`, will be stored in the `src/test/sam` directory. The arguments to `runtests` will be the names of the classes to be tested, without the `.py` suffix. Additionally, `runtests` will accept three options:

- `-u`: which ensures Unit Tests are run
- `-i`: which ensures Integration Tests are run
- `-p`: which ensures `pychecker` is run

If no options are given, the default is to run all three different types of tests for the classes given.

The script creates the Unit Testing Reports (see section 4.6) and Integration Testing Reports (see section 5.6) automatically, so all that needs to be done by the tester is to rename the reports with the correct number and move them to the specified test report directory.

1.8 Critical Units

1.8.1 SAM Critical Units

The critical classes of SAM are:

- MainWindow
- NetLogInput
- StreamsLogInput

These units must be fully-tested and will receive special attention from the testers in the form of more comprehensive unit and integration tests over and above that of the Non-Critical units.

1.8.2 FCM Critical Units

FCM consists of three modules:

- Router
- Receiver
- Logger

Each of these three modules are classed as critical, since each contribute to the full functioning of FCM. For details on the testing strategys that apply to these units, see section 13.

2 Bugs

This section outlines procedures for finding and dealing with bugs or defects.

A bug or defect is defined as an error in code that's already in the repository. It is expected that since the Unit Testing procedure says that a unit must not break a test case that had been passed in an earlier version, Unit Testing will not uncover any bugs except in the case when new tests are created. The vast majority of bugs will be found either by running through system tests, or by Informal Testing of the software as a whole.

2.1 Defect Severity

There are five classes of defect:

- A Severity 1 or Fatal defect is defined as an abnormal end to a program on startup, so that no work or testing can be carried out.
- A Severity 2 or Critical defect is defined as an abnormal end to a program, process or function resulting in corrupted data and/or production hold up for the user.
- A Severity 3 or Major defect is defined as a defect for which no acceptable workaround is available leading to significant impact for the user.
- A Severity 4 or Minor defect issue is defined as an error or bug in the system functionality that does not interrupt the operational flow due to availability of a workaround. Also defects that are a cosmetic change, which disrupts neither the functional nor operational flow to the existing functionality of the system, also fall into this category.
- A Severity 5 defect is defined as a problem that does not cause the program to break requirements. For example, the suggestion that various controls are not user-friendly would be a severity 5 defect.

2.2 Defect Reporting

When a defect is discovered, the person who found it must post to the mailing list a message with the keyword [bug] in the subject line. The post must include a description of the bug, the date the bug was found, a Defect Severity rating and a Description of the problem and impact.

When a bug has been fixed, an email must be sent in reply to the original defect report stating who fixed the bug, how the bug was fixed and the date it was fixed.

The Test Manager must note and track the bug in the bug log as described in the SQAP section 8.

2.3 Corrective Action

As already stated in the SQAP, if a bug is found, the Team member responsible for that section of code must attempt to correct the bug. For more information on corrective action, see section 8.2 of the SQAP, Corrective Action.

Once a bug has been corrected, The Test Manager must ensure that whenever possible a new test case is written that will catch any re-occurrence of that bug.

3 Informal Testing

This section describes what happens before code is ready to be committed in to the group CVS repository.

Entry criteria: Since this kind of testing is informal, it is to be carried out while the classes are being coded.

Exit criteria: The coder is confident that his or her code works as required and contains no obvious bugs.

Informal Testing is the testing carried out by a coder while working on a class. The main point of Informal Testing is to increase a coder's confidence in his or her code, and to identify any small bugs early on. As the cost of fixing bugs is smaller the earlier they are found, this step is invaluable for streamlining the testing procedure.

There is no set procedure for Informal Testing, and this is entirely up to the coder to implement. However the coder is encouraged to use the automated testing script provided which allows the coder to efficiently run both unit and integration tests, without the need to submit the test reports.

4 Unit Testing

This section outlines testing procedures that take place at the lowest level. In Python, the smallest compilable program unit is a class, therefore Unit Testing will be done on each class. Each class is to be tested separately from each other class to ensure that it works as specified in the SADD. Each class must pass all of the unit tests before it can be integrated into the system.

Unit tests for SAM are to be stored in the repository under `src/test/sam/unit`.

4.1 General Strategy

Once the author of a class has completed coding for a feature of the class, they must run all the Unit Test cases for that class. The test cases for a particular class are to be written by someone designated by the Project Manager. Wherever possible, the author of a class' Test case will be a different person than the author of the class. Also, wherever possible, the test cases for a class are to be written at the same time as the class, so that at least some of the test cases can be used by the author during Informal Testing. The class must pass all of its test cases before it is considered finished.

Unit tests must be performed in isolation, that is, a class must not be able to access any other classes in the program. Classes must be tested in isolation to ensure that each class functions independently from the others. However, some classes depend on other classes, and will not function without them. To remedy this situation, a number of stubs will be created. A stub is simply a dummy class that simulates the behavior of a class for testing purposes. These stubs will be stored in CVS under `src/test/sam/unit/stubs`. In order to unit test a class that imports another SAM class, the class must be copied into this directory before test cases are run.

Unit Tests will be written using a Black Box methodology (see section 4.2), and the particular tests will be chosen by some or all of Equivalence Partitioning (see section 4.2.1), Boundary Value Analysis (see section 4.2.2) and Random Selection (see section 4.2.3). For the more important logic classes, Glass Box testing will also be used if time permits (see section 4.3).

For classes that involve the creation and/or manipulation of a graphical user interface (GUI), Unit Testing is more difficult, since these classes rely on external user events to trigger functions, which cannot be easily reproduced. Therefore, Unit Testing will only be done on GUI classes if it is deemed suitable and helps, rather than hinders, development. For more details on how GUI classes will be tested, see section 7.

4.2 Black Box Testing

Black Box testing is the process of testing a unit against its expected behavior. That is, particular inputs should produce particular outputs. These tests are to be based on the interfaces given in the SADD. The test cases for Black Box testing are chosen by Equivalence Partitioning, Boundary Value Analysis and Random Selection [1].

4.2.1 Equivalence Partitioning

Equivalence Partitioning is based on the idea that members of a certain input domain are equivalent as far as testing goes. The input domain is first divided into partitions and then test cases are chosen to incorporate inputs from each partition. For each partition, only one valid input value is used, as it is deemed equivalent to other valid inputs in that partition. In addition to this valid input, invalid inputs are also tested.

The use of Equivalence Partitioning reduces the number of test cases required to cover the domain and therefore reduces the number of test cases needed to be written in total.

Example: For the input domain of [1,2,3,4,5,6,7], testing with the valid value 4 and the invalid values of -5 and 10 would be sufficient for Equivalence Partitioning.

4.2.2 Boundary Value Analysis

Boundary Value Analysis is to be used where an input domain is specified. Test cases will be created to test the boundary of the domain. Both boundary values are tested, as well as values directly above and below these.

The purpose of Boundary Value Analysis is to test for off-by-one or fencepost errors where the whole domain may not be covered.

Example: For the input domain of [1,2,3,4,5,6,7], it would be necessary to test with the values 0,1,2,6,7 and 8 to satisfy Boundary Value Analysis.

4.2.3 Random Selection

Where Black Box testing and Boundary Value Analysis may not be appropriate, Random Selection of data will be used. While Random Selection of data is not designed to test portions of code especially prone to bugs, it is useful for testing multiple and large amounts of test data and for generating lots of test cases.

4.3 Glass Box Testing

Glass Box testing is designed to maximize coverage of the code. Test cases for Glass Box testing need to be created to provide path coverage and branch coverage. Python has a code coverage feature as part of PyUnit which may prove useful in Glass Box testing. Since Glass Box Testing involves a lot of time and attention, it will only be used if time permits and only on the more important logic classes.

4.4 Naming Conventions for Unit Tests

Each Unit Testing file will be called UTCLASSNAME.py, where CLASSNAME is the name of the class the test is designed for. For example, the unit test file for Mean.py will be named UTMean.py.

Each individual Unit Test will be named after the function that it tests. The format of this name will be the function name, followed by the word Test, followed by the number of the test. For example, unit tests for the getName function should be called getNameTest1, getNameTest2, and so on.

4.5 Procedure for Unit Testing

Entry criteria: A unit must have undergone Informal Testing (see section 1.5) by the author. This ensures that the unit compiles and has a good chance of passing all the unit tests. It also provides a chance for the person most familiar with the code (the author) to test it.

Exit criteria: Since a class may be in a development stage during Unit Testing, it may not yet have full functionality. As such, it is not expected that a class will pass all of its unit tests before it is finished. Therefore, each class only needs to pass all unit tests that it had previously passed in order to meet the exit criteria for unit testing. That is, a test that produced a pass in a previous version of code must produce a pass in all later versions. A class cannot be considered finished however, unless it has passed all of its unit tests.

The procedure for Unit Testing is as follows:

1. At the time an author is assigned to write a class (or even before), a unit tester is also assigned to write the unit tests for that class.
2. As unit tests are finished, they are made available to the author to assist in Informal Testing.
3. The author must carry out Informal Testing on the class while it is written. The Informal Testing may make use of the Unit Tests that have been written and also any test data the author thinks appropriate.

4. Once a class has been changed and is ready for formal testing, the author must use the runtests script as described in section 1.7 to run Unit Tests on the class. The script will copy the class into the stubs directory (see 4.1) and then run all the unit tests that are available for that class.
5. The class must pass all test cases that it had previously passed (as explained above). If not, then the author must stop and fix the class (or the test case if the interface to the class has changed) to ensure this.
6. A Unit Testing Report (see section 4.6) is created by the author by running the unit tests.
7. Once the report has been created and stored under `/test-reports/sam/unit`, the changes may be checked into CVS.

Any time a class in CVS changes (source code only, not documentation changes) and is therefore re-tested (for example, during Integration or System Testing), another Unit Testing Report must be created, and the changes posted for review as above.

4.6 Unit Testing Reports

Unit Testing reports are to be named UNITNAME.UTreport.REPORTNUMBER, where UNITNAME is the name of the unit that has been tested, and REPORTNUMBER is the number of the report. These reports are to be stored under `/test-reports/sam/unit` in the home directory. For example, if unit GetFood is tested for the first time, the report will be called GetFood.UTreport.1. The format of a Unit Test report is just the output from running a PyUnit (see section 14.1) or Mtest (see section 6) test suite. Unit Test Reports are created automatically by the testing script (see section 1.7).

Here is an example of a Unit Testing report:

```
-------------------------------------------------------------
Unit Test Report for AboutDialog

Fri, 12 Jul 2002 15:11:35 +0000

-------------------------------------------------------------
Running setUp: no setUp needed for this unit test
-------------------------------------------------------------

Name: Test1
Purpose:        Tests that the widget_tree of AboutDialog is assigned
Uses:           AboutDialog
Preconditions:  None
Postconditions: None
Author:         mayadm
Date:           12/7/02

Test case Passed
.Running setUp: no setUp needed for this unit test
-------------------------------------------------------------
Name: Test2
Purpose:        Tests that the widget_tree of AboutDialog hides itself
Uses:           AboutDialog
Preconditions:  None
Postconditions: None
Author:         mayadm
```

```
Date:           12/7/02

Test case Passed
.
----------------------------------------------------------------------
Ran 2 tests in 0.109s

OK
```

5 Integration Testing

Once classes have been successfully unit tested, there must be a clear, well-defined strategy in place for combining them together incrementally to form the final product. If the whole system is simply combined together after Unit Testing (the Big Bang approach) and there is a bug, it will not be easy to find the bug. Hence, the system will be built incrementally, and tested at each stage. This is, in effect, the build plan, which has been represented here as Integration Testing, and in particular, the Order of Integration. Integration Testing is designed to ensure that the classes interface with each other correctly (as defined in the DDD), and hence is mainly concerned with the interfaces between classes (and not the internal workings of classes). Once subsystems have been built, Integration Testing also tests that each subsystem as a whole is performing its responsibilities. Before any Integration Tests are applied to classes, these classes must have passed all their Unit Tests.

As with Unit Tests, Integration Tests will be written using PyUnit (see section 14.1) or with Mtest (see section 6), and run using the runtests script (see section 1.7).

5.1 General Strategy

The chosen integration strategy for SAM is a combination of Thread Based, Bottom Up and Top Down. This section explains the rationale for choosing these methods.

5.2 Naming Conventions for Integration Tests

Each Integration Testing file will be called ITCLASSNAME.py, where CLASSNAME is the name of the class the test is designed for. For example, the integration test file for Mean.py will be named ITMean.py.

Each individual Integration Test will be named after the function that it tests. The format of this name will be the function name, followed by the word Test, followed by the number of the test. For example, integration tests for the getName function should be called getNameTest1, getNameTest2, and so on.

5.2.1 Breakdown of SAM

Firstly, the breakdown of SAM is based around the MVC architecture, as explained in section 1.2. Typically in MVC, the Controller class interacts with the user and controls the entire system, the Model module processes data and the View module displays data to the user. However, seeing as SAM must perform large amounts of analysis, it has been chosen to split up the usual Model module into two: one (also called Model) handles the loading and reading in of data, and the other (Analysis) handles the processing (analysis) of data. Also, the View module has been renamed to Output. This modified version of MVC corresponds to using SAM for doing an analysis. Also, there is an extra module not usually part of MVC - the Logger, which is responsible for creating network log files (that will be used by SAM for analysis). This extra module corresponds to a separate function SAM may perform. This breakdown can be seen in Figure 1.

5.2.2 Thread Based Integration

In the SRS and DDD, detailed use cases were developed to give examples of SAM's behavior. These use cases are invaluable, since they show the effects of one particular user event or input to SAM, which corresponds to independent ways SAM may be used - ie, one particular thread of execution. Clearly, it can be seen from the breakdown of SAM that SAM may be used for two independent functions - Logging and Performing Analysis. These two functions correspond to different sets of use cases in the SRS and SADD (as shown below), and hence are the two

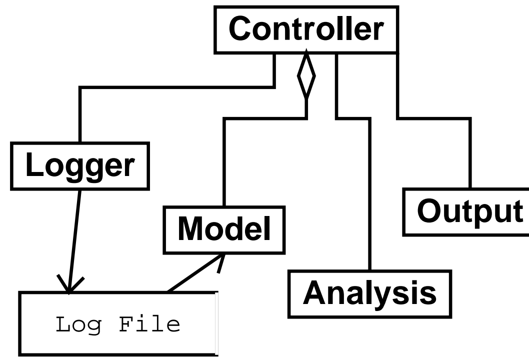

Figure 1: Shows the modified MVC architecture used for SAM, including the decomposition of the model module into Model and Analysis modules.

threads of execution.

Thread 1: Performing Analysis: 'Load File' then 'Select information relevant to analysis' then 'Perform analysis' then 'Output result of analysis'
Thread 2: Logging: 'Create Log' then 'Select information relevant to logging' then 'Do log' then'Log file created'

Thread Based Integration involves building then integrating all the classes involved in a particular thread, where each thread involves some input, some processing and some output. Here, this corresponds to the Controller and Logger modules for Thread 2, and the Controller, Model, Analysis and Output modules for Thread 1.

The advantages of Thread Based Integration are:

- Different threads of execution can be built and tested in parallel.
- It is useful for user interface and event driven systems like SAM.
- Once a thread has been successfully built and tested, it can be shown to the Client.
- Important classes can be integrated early to allow maximum testing.
- The structure of the system can be assessed early.

5.2.3 Bottom Up Integration

However, to simply choose Thread Based Integration is not enough, since it does not define the order in which each thread (and hence each class of each module) will be coded and tested, only the overall order. Therefore, another strategy needs to be chosen for coding and testing the modules (Controller, Logger, Model, Analysis and Output).

For all the modules (except Controller - see next section), the chosen strategy is Bottom Up. Within each module, the lower classes perform the main computations and calculations, therefore it is important that they get adequately tested. This also reduces the need to write stubs, since the lowest level classes (ones that do not call other classes) are coded and tested first.

5.2.4 Top Down Integration

The one exception to the Bottom Up strategy chosen is the Controller module. This module controls the entire system and directs each other module as to what to do. Clearly, it would be unacceptable to code this module last (as would be done in Bottom Up). The topmost class

of the Controller module (called MainWindow) must be in fact the first class coded, since the architecture of the system depends on this, and hence architecture flaws can be detected early. This corresponds to a Top Down strategy for the Controller module - coding the topmost class first (and writing stubs), then filling in the stubs with the lower classes.

5.3 Order of Integration

The order in which classes will be integrated is important, since it defines the order in which classes must be coded. This will prevent time being wasted in waiting for classes to be coded, and will make the testing process more efficient. **Note:** The numbers in brackets, eg, (4) or (iv) or (d) refers to the 4th step of integration for the particular order of integration of the current list.

1. **Controller module**
   The order of integrating the Controller module is as follows:
   (a) MainWindow + Selector
   (b) (a) + FileSelection
   (c) (b) + AboutDialog
   (d) (c) + AnalysisPlugins
   (e) (d) + OutputPlugins
2. **Thread 1: Performing Analysis**
   The order of integrating Thread 1 is split into integrating each module involved (Model, Analysis, Output) in a Bottom Up fashion, and then combining these modules with Controller. *Note:* Each of Model, Analysis and Output can be coded and tested in parallel.
   (a) **Model module** The order of integrating the Model module is as follows:
      i. Model + NetLogInput
      ii. (i) + StreamsLogInput
      iii. (ii) + Table

      Note: During development, it was decided to change to using Field and Stream classes for holding data about fields and streams, rather than just using lists of tuples. This resulted in two new classes being added and needing integration into the system. The order used for this was:
      i. Stream + StreamsLogInput
      ii. Model + (i)
      iii. Field + NetLogInput
      iv. (ii) + (iii)
      v. (iv) + Table
   (b) **Analysis module** The order of integrating the Analysis module is as follows, noting that the order starts with integrating the easier classes first (Plot), then moving towards the harder classes (AutoCorrelation), to enable us to ease into the harder analyses. Since the analyses modules will be plugins, they can be coded independently from other analyses, which is why the order below involves only integrating with AnalysisPlugins.
      i. AnalysisPlugins + Plot
      ii. AnalysisPlugins + Mean
      iii. AnalysisPlugins + Variance
      iv. AnalysisPlugins + PDF
      v. AnalysisPlugins + CDF
      vi. AnalysisPlugins + SpectralAnalysis
      vii. AnalysisPlugins + AutoCorrelation

(c) **Output Module** The order of integrating the Output module is as follows. The outputs are also plugins, so the same comments as per the Analysis Module above apply here as well.

    i. OutputPlugins + Graph
    ii. OutputPlugins + TableToFile
    iii. OutputPlugins + TableToScreen

(d) Controller + Model

(e) (d) + Analysis

(f) (e) + Output

3. **Thread 2: Logging**
   The order of integrating this thread is as follows:
   (a) NetLogOutput + LogFile
   (b) (1) + StreamsLogOutput
   (c) Network + NetLogger
   (d) (3) + (2)
   (e) Logger + FileChooser
   (f) (5) + SelectFields
   (g) (6) + (4)
   (h) Controller + (7)

4. At this stage, Thread 1 has been integrated and Thread 2 has been integrated. All that remains is to ensure each thread works when the other thread is present. This can be done in System Testing.

5.4 Procedure for Integration Testing

Entry criteria: Any unit about to undergo Integration Testing must have passed all its unit tests. This ensures that Integration Testing focuses on interfaces between classes or modules and not internal structure.
Exit criteria: The thread being integrated passes its Integration Tests.

The procedure for Integration Testing is as follows:

1. Once all the classes corresponding to a particular thread have been coded and unit tested, the authors of the classes will work together to come up with a set of integration test cases.
2. These test cases may be made up the unit tests for the classes at the ends of the thread. For example, if a thread is made up of modA - modB - modC, then the integration test could use the unit test input for modA and expected output of modC.
3. A tester for the thread will be allocated (via email or at a Team Meeting), and this tester is responsible for running the test cases (using the testing script as described in 1.7) and creating a report for the thread (see section 5.6).
4. Once the testing is complete, the tester must notify the authors of the thread of the outcome of the test.
5. If the test was unsuccessful, the tester must either fix the problem or request for another person to fix it.

Once this step has been performed for the first time for a particular thread, it will become necessary to run integration tests along with unit tests before committing any changes to code.

5.5 Test Case Selection

For Integration Testing, the testing method used will be Black Box testing (see section 4.2, and hence test cases selected will be black box tests. This is because internal class structure is assumed to already have been tested (in Unit Testing), and also since Integration Testing is primarily concerned with data flowing across the interfaces.

The test cases selected will contain both valid and invalid data. These will be stored in directory `src/test/sam/integration` in the group's repository.

5.6 Integration Test Reports

For each class tested, a test report shall be produced and stored in the `test-reports/sam/integration` directory of the group's home directory. Integration Test Reports are to be named CLASSNAME.ITreport.REPORTNUMBER, where CLASSNAME is the name of the thread being tested and REPORTNUMBER is the number of the report.

The format of an Integration Test report is just the output from running a PyUnit (see section 14.1) test suite. Integration Test Reports are created automatically by the testing script (see section 1.7).

5.7 Regression Testing

All classes undergoing Integration Testing are assumed to have passed their respective Unit Tests. However, during Integration, errors may be introduced into the code which cause the classes to fail their original Unit Tests. It is for this reason that regression tests will be performed after each Integration step.

To ensure regression testing is carried out adequately, nightly test runs will be performed. This will be automated, and will include running all unit and integration tests, and will automatically generate Unit and Integration Test Reports for every unit/thread tested. Once the nightly test is complete, a summary report will be emailed to the Team's mailing list. This summary report is intended to find new bugs that have been introduced - ie, if a unit/thread fails any tests that it has previously passed, this unit/thread will be included in the summary report. It is then the Test Manager's responsibility to ensure these failures are attended to.

6 Testing Procedure For Logger

The Logger component of SAM is not written in Python, but in C. Therefore it is not possible to use PyUnit in order to test it. A number of unit testing frameworks for C were considered and one called Check was chosen due to it's nice neat syntax and ability to run tests under different address spaces in order to produce a test report even when the code crashes. For information on check see `http://check.sourceforge.net/`.

Unfortunately however, due to problems installing Check, it could not be used for this project. Instead, a small testing framework will be written by the person writing the test cases for Logger. This framework will be kept in CVS under `src/test/sam/Mtest/` in the files `Mtest.c` and `Mtest.h`.

The framework will be written in the C programming language and will provide functions for

- Displaying all test output to the screen or to standard output
- Keeping track of how many tests have been run in the test suite
- Keeping track of how many of these have failed
- Keeping track of how many of these have passed

Other than this difference, Testing for SAM's Logger module will follow the normal procedures in this document for SAM.

7 GUI Testing

Entry criteria: Before GUI Testing can take place, the GUI (or relevant parts of it) must exist such that these tests are relevant.

Exit criteria: A GUI Testing report must be generated showing where the GUI functions as expected and where it does not.

Due to the nature of GUI Testing, it cannot be automated. Therefore, to test the GUI, the GUI Tester must actually run the GUI and report on its behavior. The GUI Tester must make note of the following [1]:

- Can windows be resized, moved and scrolled?
- Does the window properly regenerate when it is overwritten and then recalled?
- Are all relevant pull-down menus, tool bars, scroll bars, dialog boxes, buttons, icons and other controls available and properly displayed?
- Is the name of the window properly represented?
- Is the active window properly highlighted?
- Do multiple or incorrect mouse clicks within the window cause unexpected side effects?
- Does the window close properly?
- Do pull-down menu operations work properly?
- Are all pull-down menu functions properly listed?
- Is text typeface, size, placing and format correct?
- Does each menu function perform as advertised?
- Are the names of menu functions self-explanatory?
- Is alphanumeric data entry properly echoed?
- Do graphical modes of data entry work properly?
- Are data input messages intelligible?

7.1 GUI Test Reports

Test reports for GUI Testing will be stored in the group directory under `test-reports/sam/GUI`. These reports will be named GUITReport.NUMBER where NUMBER is the number of the report. GUI Testing is to take place on a weekly basis at the same time as the Weekly Code Review. The job of GUI Testing will rotate around the Team. The Review Manager will designate a team member to perform the GUI Testing as part of the Weekly Code Review.

The GUI test reports will also contain the System Testing report, as these steps will be performed at the same time by the person allocated to do GUI testing.

8 Function Testing

Entry criteria: Before Function Testing can take place, the system must have undergone Integration Testing, and be fully integrated.

Exit criteria: The system passes all tests.

Function Testing will be conducted to ensure the system meets the requirements set out in the SRS. A checklist will be created from the SRS detailing each requirement. This checklist will be stored in CVS in the file `src/test/function/FTChecklist`.

To perform Function Testing, the tester will step through the checklist and respond to each point with either PASS or FAIL. If the tester's response is FAIL, any extra comments may be added. This altered checklist file will serve as the test report for Function testing.

Test reports for Function Testing will be stored in the group directory under `test-reports/function`

8.1 Function Test Reports

For each function test, a test report must be created. The test report must contain, in addition to the updated checklist:

- The name of the test
- The username of the tester
- The date the test was run
- The result of the test (Pass or Fail)
- Any further comments

9 System Testing

During Unit and Integration Testing, the focus was on correctness of code and catching bugs. During System Testing, however, the aim is to ensure that the system, once fully integrated, meets the standards and objectives that were initially specified in the SRS. System Testing involves testing both the functional and non-functional requirements of the system as set out in the SRS.

System Testing will be conducted on a weekly basis by the GUI tester (see section 7). To perform System Testing, the tester must go through the text file `usecases.txt` and ensure that SAM gives the correct responses to the input.

The `usecases.txt` file is to be stored in the repository under `src/test/GUI`.

9.1 General Strategy

These tests will be conducted only when all parts of the software have been integrated into a whole.

9.2 System Test Reports

For each system test, a test report must be created. The test report must contain:

- The name of the test
- The username of the tester
- The date the test was run
- A description of the test
- The test data
- The expected output
- The actual output
- The result of the test (Pass or Fail)
- Any further comments

The format of System test reports should follow that of `usecases.txt`. System test reports will be stored within the GUI test report (see Section 7.1) for that week under the heading of Usecases.

10 Installation Testing

Entry criteria: Before Installation Testing can take place the system must have undergone Function Testing, and be fully integrated.

Exit criteria: Installation of the system must be successful.

Installation Testing is conducted to verify that the software can be installed on the hardware/software platform specified in the SRS. Test cases for Installation Testing will follow through the procedure in the installation manual to install the software. As such, Installation Testing will also validate the installation guide.

If possible, Installation Testing will also take place on a machine matching the minimum hardware and software requirements criteria in the SRS.

10.1 Installation Test Reports

Each time Installation Testing is performed, a test report must be created. The test report must contain:

- The name of the test
- The username of the tester
- The date the test was run
- A description of the test
- The test data
- The expected output
- The actual output
- The result of the test (Pass or Fail)
- Any further comments

Test reports for Installation Testing will be stored in the group directory under `test-reports/instest` and will be named InsTReport.NUMBER where NUMBER is the number of the report.

11 Documentation Testing

Entry criteria: Before Documentation Testing can take place the system must have undergone Function Testing, Installation Testing and be fully integrated. Also, the Documentation must have been written.

Exit criteria: The process outlined below must be carried out, and a test report showing the documentation to be correct must have been generated.

Documentation Testing is performed by going through the documentation and performing each step relating to the use of the program. The Documentation Tester must examine the documentation for accuracy and verify its contents against the program. The Documentation Tester must also examine the Documentation for editorial clarity - that is correct spelling, grammar and clarity of instructions. During the test, the Documentation tester must make note of the following in the test report [1]:

- Is it easy to locate guidance within the document?
- Are the document table of contents and index accurate and complete?
- Are all error messages that may be displayed to the user described in the documentation?
- Are all the tool-tips or other in-program documentation accurate?

Some sections of the Documentation may have already been tested during Installation Testing (see section 10) or GUI Testing (see section 7) - these need only be performed once on the same version of code, and as such may be exempt if already listed in an Installation Test report or GUI test Report, however a note must be made in the Documentation Test report that these tests have already been done. Additionally, the user documentation will undergo a review process as explained in the SQAP in the Reviews section. This review will also help to ensure the correctness of the user documentation.

12 Acceptance Testing

Entry criteria: All other forms of testing have been performed.

Exit criteria: All Acceptance Tests pass.

Acceptance Testing is conducted to show the Client that the system performs as specified in the SRS. Acceptance Testing will consist of following through a number of the core requirements set out in the SRS to demonstrate the system's functionality.

Acceptance Tests must be clearly derived from the SRS and signed off by the client to ensure that passing the acceptance Tests means the product has satisfied the SRS.

Final tests will take place at the Client's site once RAMI is complete.

Test reports for Acceptance Testing will be stored in the group directory under `/test-reports/sam/Acceptance` for SAM and `/test-reports/fcm/Acceptance` for FCM.

13 Testing procedures for FCM

This section describes the approach and procedures to be used when testing FCM. This includes the two kernel modules, and the logger component.

13.1 Stress Testing

Due to the complex nature of the Linux kernel, it is too difficult to create drivers and stubs for all the resources the kernel modules use. Hence, apart from compilation, all testing of the two kernel modules will take the form of stress testing. Stress testing will apply loads to the network and ensure that this does not cause a failure of the kernel modules. Varying lengths of time will be used, depending on the level of testing being performed.

13.1.1 Precheckin Testing

Before changes can be made to the source code of the FCM kernel modules, the modules must undergo a short period of stress testing. As a compromise between rigorous testing and the need to not hinder changes, a duration of 5 minutes has been chosen. The standard test load will be applied during this time. If the module is still operating as expected at the end of testing, it is considered to have passed. If however the network no longer works, or the kernel has crashed, it is considered to have failed. In this case, the change must not under any circumstances be committed.

13.1.2 Nightly Testing

As both an indicator of progress, and to ensure new fatal bugs are not introduced, the test load will be run each night. This is to be run for a duration of 1 hour. The same criteria exist for determining success or failure as in Precheckin Testing (see section 13.1.1).

13.1.3 Standard Test Load

To maintain some level of reproducibility, a standard set of network connections will take place as part of the test load. The following types of connections will occur:

1. **Standard TCP connections** - A series of large HTTP transfers from hosts on the local network will take place. Some of these will run concurrently. The contents of the file being transferred is not important, so a random file, created from `/dev/urandom` will be used. A 50 Megabyte file will be used for this. It can be created with the following command:
   `dd if=/dev/urandom of=testfile bs=1k count=50k`

   This file will then be accessible from an HTTP server located on a host running the Router module. The `wget` utility will be used to download this file from a host running the Router module.
2. **Corrupted TCP packets** - To ensure FCM does not crash when corrupted TCP packets are received, some will be sent on the network.
3. **Non TCP connections** - To ensure FCM handles non-TCP packets correctly, some ICMP and UDP packets will be sent. These will be sent from both the Router and the Receiver, and directed at each other.

These connections will be repeated until the desired duration is reached. It is important to note that these tests are primarily for stability and error handling.

13.2 Correctness Testing

In addition to the above tests, some basic unit tests must be run before committing changes to FCM. These will include the following tests:

1. **Compilation -** Check that the module compiles successfully
2. **Loading -** Check that the module successfully loads and unloads into the current Linux kernel

These tests are automated via the `runtests` script, which is located in the `src/test/fcm` directory of the CVS repository. It produces a report giving details of the tests.

13.3 System Testing

FCM will also undergo System Testing, see section 9. This will take place once the Test Manager in conjunction with the coders of FCM feel FCM has met the requirements as specified in the SRS. After which System Testing will be conducted at the Test Manager's discretion.

This will ensure that the FCM modules are not only efficient and bug free but also follow what the client wanted as specified in the SRS.

13.4 Installation Testing

FCM must also undergo Installation Testing. See section 10 for details.

13.5 Acceptance Testing

FCM must also undergo Acceptance Testing. See section 12 for details.

14 Appendices

14.1 PyUnit

PyUnit is a testing framework for the programming language Python. It will be used mainly in the writing of unit and integration tests, but may also be used during other testing phases.

A PyUnit test is made up of a class that inherits from unittest.TestCase. These tests are combined into test suites which are derived from the unittest.TestSuite class.

Once a Test Suite has been created, a runner object, of type unittest.TextTestRunner is created to actually run all the tests in the suite. The runner outputs the results of the test suite. Information on PyUnit can be found here: `http://pyunit.sourceforge.net/`

The template to be used for PyUnit files is as follows:

```
#
# File: UTTemplate.py
#
# vim: ts=4 sw=4 noexpandtab:
#
from time import localtime, strftime
import sys

sys.path.append("../../../sam/")

import unittest
import someclass

hline = "-------------------------------------------------------------\n"

class someclassTestCase(unittest.TestCase):
"""One line summary of the class (this must be on one line!!).
Purpose:    A more detailed description of the class and its
intended behaviour.
Uses:       Class1, Class2
Author:     login
Date:       13/5/02
"""
# Setup Function
def setUp(self):
"""Purpose:        Sets up X for testing by doing
blah and doing blah.
Uses:           Class2
Preconditions:  Assumptions about the input
Postconditions: Assumptions about the output
Author:         login
Date:           13/5/02
"""
sys.stderr.write(hline)
sys.stderr.write("setUp finished\n")
return

def Test1(self):
"""Name: Test1
Purpose:        What the test is expected to test
```

```
Uses:           Class2
Preconditions:  Assumptions about the input
Postconditions: Assumptions about the output
Author:         login
Date:           13/5/02
"""
sys.stderr.write(hline)
sys.stderr.write(self.Test1.__doc__)
# testing & assert goes here
sys.stderr.write("Test case Passed\n")
return

# The code to run the Test Suite

# print header
sys.stderr.write(hline)
sys.stderr.write("Unit Test Report for Someclass\n")
sys.stderr.write(someclassTestCase.__doc__)
sys.stderr.write(strftime("\n%a, %d %b %Y %H:%M:%S +0000\n", localtime()))
sys.stderr.write(hline)

# Set up the suite
someclassTestSuite = unittest.TestSuite()
someclassTestSuite.addTest(someclassTestCase("Test1"))
# Run the suite
runner = unittest.TextTestRunner()
runner.run(someclassTestSuite)
```

References

[1] Roger S. Pressman. *Software Engineering: a Practitioner's Approach*. McGraw Hill, fourth edition, 1997.