CHALMERS

# Automation of Negative Testing

Master of Science Thesis in the Programme Software Engineering and Technology

GODWIN SEBABI SEMWEZI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden,  June 2012

Automation of Negative Testing

GODWIN SEBABI SEMWEZI

Examiner: SVEN ARNE ANDREASSON
Supervisor: ROBERT FELDT

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2012

## Acknowledgements

I would like to thank my supervisors Associate Professor Robert Feldt and Dr. Sigrid Eldh for the support and guidance they provided throughout this study. This thesis would not have been possible without their valuable time. I am thankful for the numerous meetings, discussions and feedback that we had, which were a great inspiration and helped me to focus on the subject and most important issues.

## Preface

This Master of Science Thesis is reported here in a hybrid format, i.e.the main content of the Work is reported as a scientific article conforming to the Empirical Software Engineering Journal’s template, complemented by additional appendices relevant for the examinationat Chalmers University of Technology.

Table of Contents

## Contents

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF SCREEN SHOTS

# Automation of Negative Testing

Godwin Sebabi Semwezi

**Abstract**

Software testing primarily focuses on ensuring the right system behavior when the expected inputs are provided. As software becomes more complex it presents numerous use cases that are prone to failures. Testing must explore these use cases to ensure appropriate system behavior in the presence of faults, an endeavor often called Negative Testing. However the knowledge on Negative Testing is limited which hinders its application to testing. In this study, the existing knowledge and techniques for Negative Testing is explored resulting in a new perspective of Negative Testing. We call this Negation Testing since the previous term is broad and has been used in many different meanings. This new perspective is then implemented as an extension to a unit testing framework. The extension is evaluated and results show that the presented approach can reveal faults and unexpected behavior in software.

## 1 Introduction

Developing quality products is important for an organization to be successful in the new global economy [1]. Increased software complexity, competition, customer expectations and improved internet access have brought the concept of quality to the forefront [1]. Software testing is one activity that plays a key role in achieving quality software.

In practice, testing focuses on providing the expected inputs to software that will ensure the right system behavior. This is known as positive testing. However, positive testing is insufficient when software controls critical systems. Examples of critical systems include business critical applications like financial transaction systems, embedded systems like fly by wire systems or a pacemaker, and cyber physical systems like remote surgery or co-operative active safety systems for road vehicles. Such systems present numerous use cases resulting in failures that are a risk to human life. Such use cases may not be covered by positive testing. Therefore it is increasingly important to also ensure stable behavior even for use cases that do not ensure the expected system behavior.

Negative Testing (NT) seeks to provide strategies and techniques to identify and check unexplored or unexpected use cases and behavior of software. However, Negative Testing (NT) is not clearly defined and understood despite its importance. This can be attributed to the multiple definitions that describe NT in different ways. The definitions focus on one or more but not all possible aspects that can be used to test systems such as inputs or the context of execution of a system. Furthermore the definitions do not guide testers on how aspects of testing can be derived from system descriptions and some of the definitions introduce subjectivity issues by stating that aspects of NT have correct and incorrect values.

In addition, recent research in Swedish software industry has shown that more research is needed to create more generally useful NT techniques [2]. One notable issue with NT techniques is that some of them are categorized as positive testing techniques which make it challenging for a tester to know when positive or NT is being carried out as there is no clear definition of NT.

This paper presents and describes some but not all of the different existing NT definitions and techniques. The paper presents a new perspective of NT as Negation Testing in which a definition and description on how the new perspective resolves the gaps identified in current definitions of NT is provided. The design of an extension of a unit testing framework in which Negation Testing is implemented is then presented and is evaluated and the results discussed.

This paper is structured as follows: Section 2 gives background information on selected NT definitions. Section 3 describes the research methods used in the study and the research questions guiding the study. Section 4 presents the related work. Section 5 describes NT techniques. Section 6 presents a new perspective of NT as Negation Testing (NT'). Section 7 describes the design of the NT extension of a unit testing framework. Section 8 presents the results of the evaluation of the testing extension. Section 9 presents a discussion of the results and the study and Section 10 concludes the paper.

## 2 Background

Negative Testing (NT) is not clearly defined and is often referred to with different names. In this subsection, selected definitions of NT are described to give a background on how NT is currently being executed. The definitions are named according to the main aspect that is explored by NT. The definitions were obtained as a result of a literature study of various sources that are referenced besides each definition.

### 2.1 Input oriented

**Testing to determine the response of the system outside of what is defined. It is designed to determine if the system doesn't crash with unexpected input [3].**

This definition describes the NT as an activity that focuses on supplying unexpected inputs to the System Under Test (SUT) and observing the subsequent behavior of the system. The aim is to verify that the SUT continues functioning and responds appropriately in the presence of unexpected inputs. The unexpected inputs have a low probability of causing the system to exhibit the expected behavior. In contrast, expected inputs have a high probability of causing the software to exhibit the expected behavior.

Software must be able to handle unexpected inputs and NT seeks to ensure that error handling mechanisms of the SUT keep the system available while sending the appropriate responses to end users. NT test cases can easily be defined by using inputs derived from the negated specifications of the SUT.

However, restricting NT to unexpected inputs leaves software open to exploitation using expected inputs. This is because systems exist in uncertain environments which can cause expected inputs to bring about unexpected behavior. For example, given that the SUT depends on services of other systems like databases, expected inputs can bring about unexpected behavior when the systems they depend on are unavailable. A soft error, which is a term used to describe an event when a particle strike alters binary information in a circuit is another example. The soft error can alter the value of expected input bringing about unexpected behavior. To recover from a soft error, the correct bit values need to be reloaded into the affected memory locations. In this case, NT would test that the correct bits are reloaded when a soft error occurs.

In Addition, the notion of what is viewed as unexpected input is subjective to the tester i.e. two people can have different views on what input is expected or unexpected. This can lead to endless debates as to which inputs should be used for NT.

## 2.2 Path oriented

**Testing aimed to identify off paths, exception conditions and other anomalous situations [4].**

Path testing is a structural testing method that involves using the source code of a program to attempt to find every possible executable path [4].

Following this definition, NT focuses on the path of execution that results in unexpected system behavior, also called the abnormal path of execution. The paths of execution can be derived from the negated specifications about inputs.

While investigating the abnormal path is important to execute NT, it restricts NT from the normal path which is subject to exploitation. There can be numerous paths of execution in software and it is difficult to verify all paths. Therefore one cannot say that what is referred to as the normal path cannot be manipulated to produce unexpected system behavior. The normal path should be investigated when executing NT.

Another issue is that testing the path of execution of software often requires testers to create situations that software was not designed to be used and is sometimes called Creative or Dirty Testing [4]. An example of such a test is testing if it is possible to watch a video file in a text editor. Such unrealistic tests have a negative impact on the overall cost of the software and the testing effort as they do not contribute to the improvement of the software. In addition, highly skilled testers are required to discover such anomalous failure situations further increasing the cost of the testing effort.

Furthermore, the identification of the normal and abnormal path of execution is subjective to the tester as different testers can have differing views on what path is normal or abnormal.

## 2.3 Specification oriented

**Negative testing is a test technique that aims to target execution paths and input, outside what is clearly defined in the specification of the system [2].**

This definition describes the NT as testing use cases of software that are not explicitly stated in the specification of the SUT.

By testing use cases that do not exist in the specification, testers do not duplicate tests that already exist reducing on the cost of testing. No restriction is put on the use of valid or invalid data, or paths of execution as is with definitions previously described above.

However, it is not clear as to what constitutes a negative test with regards to time e.g. If a negative test reveals a bug in software that is subsequently resolved and added to the specification of the software, the test seizes to be a negative test and becomes a positive test. There is no fine line between what constitutes a positive test and a negative test apart from the item being tested in the former existing in the specification and the latter being unknown. Perhaps a stronger differentiator between positive and negative testing is required other than existence in the specification such as a relation of the test to the goals or primary functions of the software.

## 2.4 Goal oriented

**“Testing aimed at showing software does not work [4]”.**

Human beings tend to be highly goal oriented and thus establishing a good goal has a psychological effect on testing [5].

The definition above aims to ascertain that software does not do what it is supposed to do. As a result the goal of NT is to focus on only those test cases that result in unexpected system behavior.

This goal is incomplete because it leaves out the test cases that exhibit expected system behavior. Test cases that exhibit expected system behavior can be manipulated in various ways such as changing the context of execution to produce unexpected system behavior.

Furthermore, test cases that exhibit expected system behavior are derived from the system specification which is often incomplete. This means that there can be some functionality of the system that is not yet explored and can cause failures.

Therefore it is important that the definition of NT has a goal that allows investigation of both expected and unexpected system behavior.

### 2.5 Discussion

The definitions of NT above show that NT can be carried out in a number of ways. But the definitions introduce aspects to be considered when executing NT.

The definitions described in subsections 2.1 – 2.3 introduced inputs as an important aspect to be considered when executing NT. The path of execution is dependent on the inputs supplied to the SUT and hence not considered a concrete aspect of NT.

The context of execution or the environment of execution of the SUT is another important aspect. Subsection 2.1 describes its manipulation as a cause of failures of the SUT. Hardware, software or humans that communicate with the system need to be understood and the ways in which communication occurs as software miscommunication with its environment is a common cause of failure. One major reason why testing the environment is neglected is that it is difficult to replicate the environment in which software will be executed by users [6].

The definition in subsection 2.4 is a goal oriented definition and emphasizes the use of a good goal for executing NT. In addition, subsection 2.2 described a good goal as being a way to avoid the design of test cases that target situations that the SUT was not created to handle as such test cases do not contribute to the testing effort.

The characteristics about inputs, context of execution and the goal can be derived from the specification, tacit knowledge or from users passed experience of the SUT.

These three aspects are used to analyze the NT techniques in the sections that follow.

## 3 Methodology

The main research method used is the Design research method described in [7] which is shown in Figure 1. The sub sections that follow briefly describe each step which is followed by the research questions that the study will answer.

### 3.1 Design Research Steps.

Awareness of the problem is the first step in the Design Research methodology in which knowledge about a problem is acquired by studying existing literature or new developments from industry related to the study. The output of the step is the proposal of the study that includes various factors that are used to evaluate the solution.

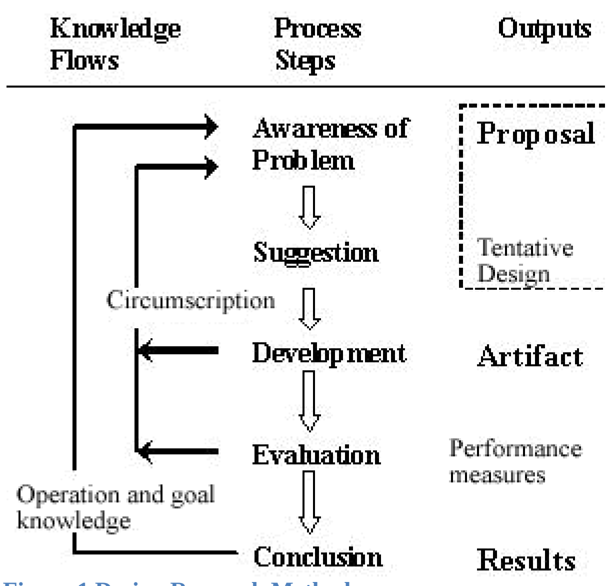


Figure 1 Design Research Method

Suggestion is the next step which involves the use of both knowledge acquired from the awareness step and new innovative ideas to develop a tentative design of the envisioned solution. For this study, an extension of a unit testing framework is designed in which selected negative testing techniques are incorporated.

The Development step follows in which the design from the suggestion step is implemented resulting in an artifact. For this study an extension for the Ruby MiniTest unit testing framework is developed.

The artifact is then evaluated and its performance based on the factors identified in the Awareness step is documented. Both qualitative and quantitative data is documented which is followed by an in depth analysis of the results resulting in new hypotheses or explanations to existing hypotheses which may serve as input to the awareness step. This may indicate a need for further research of the study. For this study, the unit testing extension is used to test various open source libraries of different complexities and the type and number of discovered failures documented. The study is then concluded with a summary of the findings and lessons learned from the study.

### 3.2 Research Questions

The main research questions guiding the study include

1. What are the negative testing techniques?
2. How can the NT techniques be automated?
3. Would the automated techniques be helpful in finding unexplored use cases and behavior in a real system?

These questions guide the exploration of the existing knowledge of NT and creation of ideas that result in the design and implementation of an extension of a unit testing framework. This extension is then evaluated and results obtained from which the main finding is that the automated NT techniques are helpful if finding unexplored use cases and behavior of systems.

## 4 Related Work

Eldh carried out a study on an operation and maintenance interface of a telecom middleware platform with the aim of evaluating the efficiency, effectiveness and applicability of negative testing on software systems [2]. The study was conducted as a part of a thesis, performed by two master students. The testing techniques used included positive tests, equivalence partitioning, boundary value analysis, fault injection, random input variation and the software attack techniques as described in [6]. It was concluded that it is difficult to apply the test techniques for at least some specific types of software in industry. One reason identified for this was the lack of time and motivation of testers to gain a deep understanding necessary for constructing and challenging the system. Another reason was that some of the techniques are not simple to translate to industrial systems.

However it is not clear as to whether the possible failure to translate the techniques was due to the complexity of the techniques or the inability of the masters' students to understand the techniques.

James describes his experience in using negative testing to test applications [8]. He discusses its management, techniques used to select, derive and execute the negative tests. James recognizes the need for experienced testers in order to carry out effective negative testing and that NT can reveal information about the risk model and increase confidence of the quality of the system.

NT is not yet fully explored and this can be seen from the limited related work. Much more research is needed to create more generally useful NT techniques. From literature we identify the following issues with some NT techniques:

- The techniques are not easy to understand.
- The techniques are not easy to implement often requiring a lot of time and skill to create scenarios that reveal unexplored use cases. In addition, some techniques are specific to a given interface.
- The techniques seek extravagant situations for which software may never be used.

Essential in the development of the techniques is to make them directly actionable and useable. This can take the form of detailed technique descriptions and examples but ideally the techniques can be supported with automation and tools. This study will describe NT techniques, some of which will be automated and tested.

# 5 Negative Testing and Test Techniques

This section provides a brief description of existing techniques that can be categorized as NT testing techniques.

## 5.1 Software Fault Injection

This is a technique in which program behavior is influenced by modifying exception or error handling code paths of the SUT in order to change its context of execution. The technique is also executed by introducing faulty inputs into software at runtime.

The goal of the technique is to allow for the evaluation of fault handling procedures, assessment of the quality of exception handling and dependability procedures [9]. The technique tries to match the SUTs responses with the specifications in the presence of specified faults with the aim of obtaining a high coverage of the configurations of the SUT [10].

The techniques key strength lies in the fact that it targets fault and exception handling routines that are rarely tested using positive testing.

However the technique requires the modification of SUT code. This means that the software executed during the tests is different from the software run by the end users and hence is not a copy of the user context. In addition, any errors within the introduced fault code can lead to incorrect test results and conclusions about the SUT.

## 5.2 Mutation Testing

Mutation Testing is a fault-based testing technique in which faults are introduced into software by modifying the source code. The modified software is then executed against a test set to verify that the faults are detected.

The goal of the technique is to help testers develop effective tests or identify weaknesses in the test data used for testing or used in sections of the code that are rarely accessed during execution [11].

One issue with mutation testing is that any statement within the SUT can be modified which becomes impractical for large software due to the high computational cost of executing the enormous number of modifications against a test set.

In addition, mutation testing requires the source code to be modified. This means that the conditions under which the software is executed during the test are not the same as when the software is executed by the end user and hence is not a copy of the user context.

### 5.3 Apply inputs that force all error messages to occur

This technique focuses on testing the SUT with inputs that will reveal defined error responses. The goal of the technique is to test error handling procedures which are often difficult to develop correctly to handle various error situations.

However the technique is limited by the fact that it doesn't account for situations in which error messages are handled at another layer of abstraction [6]. In this case subjecting a component to a series of tests will result in numerous failures. Therefore, the tester must fully understand the implementation of the application to effectively use this technique.

### 5.4 Explore allowable character sets and data types

The goal of the technique is to evaluate the robustness of the SUT in the presence of reserved inputs of character sets and various data types.

Different operating systems and programming languages based on either Ascii of Unicode character sets have reserved control structures or symbols that the SUT must be able to handle. For example the C language has the ++ symbol for incrementing numerical values. The SUT must be able to validate such inputs and return the appropriate error responses.

In addition, the SUT may be tested with various data types offered by the programming language in which case the SUT must be able to validate the invalid data types.

Table 2.2 on page 29-33 in [6] provides a guide for testers to use in executing the technique. The technique is efficient in revealing validation errors. However, numerous errors will be found if validation is done at a higher level of abstraction. .

### 5.5 Find input values that may interact and test combinations of their values

This technique involves testing the SUT with combinations of inputs that are related or are involved in the same operations. The goal is to identify inputs that affect each other in computations or utilize the same resources and test different combinations of these inputs so as to ascertain that the SUT functions for all possible input combinations.

The values in a relationship should describe aspects of common resources that can be internal or external. In addition the values may be used in the same computations [6].

The major drawback of the technique is that it is not possible to test all input combinations. A possible solution would be selecting a representative value from different partitions identified using the equivalence partition method.

### 5.6 Repeat the same input or series of inputs numerous times

The technique tests whether an application has error handling procedures for situations in which the context of execution is modified as a result of all resources that the SUT is dependent on being consumed [6].

Continuous repetition of the same inputs uses up system resources such as memory buffers, data storage space or communication with remote resources. In addition, if an application uses a remote resource, it is difficult for the application under test to know of the remote systems limitations and hence developers must have proper mechanism to prevent the system under test from crashing.

The technique is good for stress testing the application under test and can also benefit from automation as it involves executing the same task repeatedly [2].

However it may not be possible to execute this test if test resources are limited or not available

### 5.7 Force a data structure to store too many or too few values

This technique checks whether appropriate controls and error handling procedures on data structures or resources are in place so as to avoid underflow, over flow or corruption of data [6]. The test is executed by providing too many, too few or invalid inputs to the SUT.

The technique reveals errors brought about by the developers neglect of the limitations of the different data structures or resources they use. The technique requires in depth knowledge about the SUT so as to effectively test the limits of the data structures and resources.

However, it is difficult to simulate the limits of some data structures and resources which hinders the effectiveness of this technique.

### 5.8 Force the media to be busy or unavailable

This technique tests the SUT to verify that it operates appropriately during error conditions related to media. The technique verifies that the applications return the correct error codes associated with different kinds of media problems.

The drawback with this technique is that the test scenarios are difficult to simulate often requiring more resources than is available to testers. In addition, some applications have concurrent processes and it may be difficult to simulate a scenario in which different processes access the same resource at the same time.

### 5.9 Equivalence Partitioning

This technique divides software inputs into partitions from which test cases can be derived to uncover classes of errors. A single value from a partition will be treated similarly by the component producing the same system behavior and thus can be used to represent the entire class [12].

This technique can be used for both positive and negative testing. When used for negative testing, the focus is on those classes of inputs that cause the system to exhibit unexpected system behavior.

A key strength of the technique is that there is little training required and the tester does not need to understand the implementation. This is due to the fact that the classes are derived from the specification of the application that contains characteristics usually referring to a single parameter.

However, it is often the case that specifications are incomplete and therefore not all classes can be explored. Furthermore, in some situations the tester will only have access to the software's executable and general knowledge of the functions of software, but not the specification. In this case, identification of the partitions is more of guess work.

## 5.10 Boundary value analysis

This technique examines the boundary values of the characteristics or properties of software as more errors tend to occur at the boundaries. The technique is often considered an extension of equivalence partitioning in which the boundaries values of the partitions are the major areas of interest.

The technique requires that the lower and upper boundaries of a property or characteristic be examined. For each boundary, a value at the boundary, one value below the boundary and one value above the boundary is required. The technique can be used for both positive and negative testing. When used for negative testing, values below or above the boundary is of interest.

Boundary Value Analysis is valuable technique that can reveal errors if used correctly. It is easy to execute, but the major difficulty is in identifying the boundary conditions. This is because for some inputs, boundaries cannot be defined. However if used correctly, the technique can discover boundaries overlooked by programmers, discover boundaries resulting from interactions amongst subsystems and validate requirements.

## 5.11 Discussion

The techniques in subsections 5.1 to 5.10 describe different ways of carrying out NT and Table 1 categorizes the techniques in terms of the NT aspects derived from the definitions of NT in section 2. Table 1 shows the aspects explored by the different techniques. Because of this one can conclude that the techniques are related and are correctly classified as NT techniques

But, it is not sufficient to conclude that they are NT techniques. This is because the definitions in section 2 are not exhaustive and there could be other definitions that were not encountered by the author and these definitions might explore the same, different or exclude certain aspects for NT.A general definition of NT that is not a combination of other definitions is required.

Another issue is that the techniques are executed in different ways and the definitions of NT do not inform a user on how to use the different aspects of NT to derive NT test cases. For example, fault injection described in subsection 5.1 manipulates inputs and the context of execution. But which inputs or which context or code should be manipulated and how should they be manipulated to derive an NT test case. The definition should inform a tester how to derive NT test cases using any technique.

Sections 5.9 and 5.10 described equivalence partitioning and boundary value analysis as techniques that can be used for both positive testing and NT. This means that both of these techniques are hybrid techniques of testing and cannot be classified as positive or negative. One solution would be to state that a NT approach was applied to the techniques to derive NT test cases. Therefore NT should be defined as an approach to testing [8] or as a way of thinking to derive test cases.

| **Technique Reference** | **Inputs** | **Execution Context** |
|---|---|---|
| 2.1 | | X |
| 2.2 | | X |
| 2.3 | X | |
| 2.4 | X | |
| 2.5 | X | |
| 2.6 | X | X |
| 2.7 | X | X |
| 2.8 | | X |
| 2.9 | X | |
| 2.10 | X | |

Table 1: Classification of NT Techniques

The section that follows presents a new perspective of NT in an attempt to provide a general definition that can fill the gaps identified above.

## 6 New Perspective of Negative Testing

This section provides a different view of Negative testing in an attempt to fill the gaps identified in section 5.11.

### 6.1 Negation Testing

Several definitions of Negative Testing exist and different groups of individuals have different views of what it entails. Therefore a new term, "Negation Testing" (NT') is used so as to facilitate a uniform view of Negative Testing. Negation Testing (NT') is defined as:

"Negation Testing (NT') is an approach to testing that states a current set of assumptions about the system and/or the testing and then considers which tests should be done if one or more of the assumptions are negated, i.e. no longer fulfilled."

Software specification, design documents and user tacit knowledge are sources of information about the SUT from which a tester can obtain assumptions about the SUT. Negating these assumptions reveals new information about the system which can be used to develop tests for negative testing.

Figure 2 below shows the different areas of interest of negation testing and is used to describe negation testing

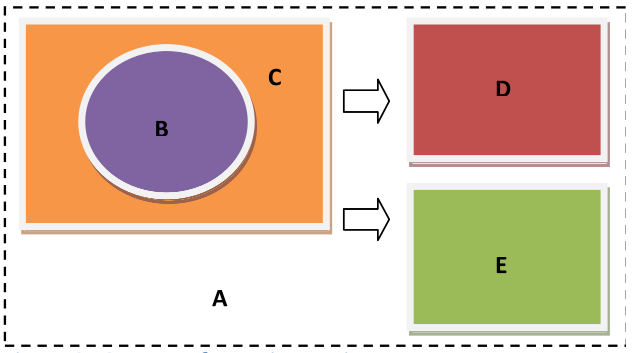


Figure 2: Context of negation testing

Below is a description of each area of interest:

- Area A represents a universe of all information about the SUT both known and unknown to the tester.
- Area B represents the known information about the SUT such as the information in software specification documents. From this, a tester can derive assumptions about the SUT.
- Areas C, D and E represent different sub-universes for which the tester challenges the SUT to acquire new information.

By negating the assumptions of area B, the tester can create test cases to challenge the SUT and acquire new knowledge about the SUT that constitutes area C. This new knowledge represents new assumptions that are added to area B. Subsequently, the new sets of assumptions of B are negated in relation to D or E to derive test cases.

For example, for a web form with a single input field that takes values from 1-10, area B includes the assumption that the field only accepts values from 1-10. Negating this assumption by supplying a value in the ranges x<1 and x>10 would constitute area C. The assumptions about the SUT are then updated with the new assumptions derived from negation tests of area C. Testing the application with a character data type and alphanumeric input would constitute the new universes, area D and E respectively. The Figure 3 below illustrates the scenario described above

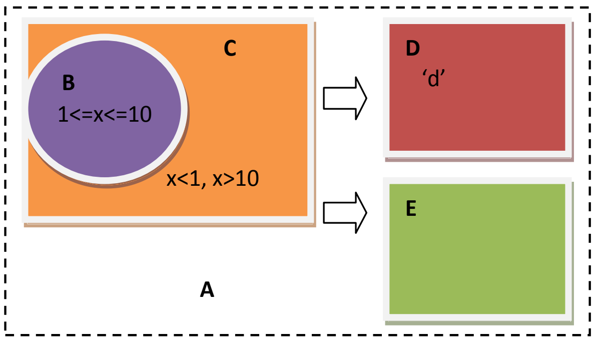


Figure 3: Negation testing applied to Web Form example

It is important to note the following about negation testing:

- If no assumptions are specified about a system, Negation Testing (NT') cannot be clearly defined as there will be no item for negation.
- NT' cannot be defined as a technique, but as an approach to testing that may be applied to any testing technique.
- A test case cannot be negative as NT' is a way of thinking to arrive at a given test case.
- The assumptions of the system are derived from characteristics of system input, context, use cases, user behavior, system behavior and tacit or knowledge about the system.
- The sub universes selected should result in test cases that test the use cases of the software and not an aspect for which the software was not intended. The goal of the software should be considered as a constraint when deriving sub universes. That is to say, the sub universes selected should be within the boundaries of the goal. For example, if the goal of a text editor is to edit text, the sub universes should be related to textual aspects of the software and not try to test the editor with media files

The negation approach means considering the countermand or reverse of what is already specified. Therefore, for each candidate testing technique, the assumptions that can be negated are identified. These assumptions are then used to derive test cases.

## 6.2 Negation Testing and Test Techniques

This section briefly describes how the NT' approach is or may be applied to two of the techniques mentioned in section 4.

### 6.2.1 Equivalence Partitioning

Equivalence Partitioning (EP) is a test design technique in which the input domain of a program is divided into a finite number of input sets or classes. A representative value of a class of input set is then tested and the behavior or result observer is assumed to be the same for any other input in the class or set. [5].

Equivalence Partitioning (EP) makes the following assumptions that can be negated to apply NT'

i. **Assumptions derived from input conditions**

In EP, classes of inputs are derived from conditions or characteristics of inputs. To apply NT', the input conditions which are the assumptions about the inputs are negated to get the invalid input classes. For example, a web form that takes input in the range 1-10, negating this condition would result in two classes of invalid inputs i.e. $x<1$ and $x > 10$.

ii. **Assumption that a test on a representative value of a given class is equivalent to any other value in that class**

EP assumes that a test on a representative value of a given class is equivalent to any other value in that class. Negating this assumption implies that not all values in a class will produce the same behavior from an application. This can stimulate identification of universes to test or use of other techniques to test the classes of inputs.

Building on the web form example above, a java version of the application returns the absolute value of the integer entered in the input field. Assuming the application correctly handles inputs in the $x < 1$ class, the application should be able to return a positive value for-2147483648; otherwise a negative value is returned which would be an invalid output. This is because the java absolute function returns a negative value for -2147483648.

### 6.2.2 Fault Injection

Fault Injection is a technique that validates the dependability of systems in which controlled experiments are executed and the systems behavior observed in presence of faults explicitly introduced into the system [10]. According to Arlat [13] fault injection has two main goals:

i. Validation of fault handling methods and mechanisms with respect to the inputs they have been designed to cope with.
ii. Support system design by applying the negative results of fault injection to initiate feedback loops to improve the test procedures and fault tolerance mechanisms.

From the above description, the assumptions below are derived which in turn are negated when the technique is executed:

**i. Assumptions about the inputs of the system under test.**

One of the goals of fault injection seeks to test fault handling methods and mechanisms with respect to inputs they were designed to cope with. This means that assumptions about valid inputs are known. These assumptions are then negated to identify invalid inputs. These invalid inputs are then injected into the system to test the error handling mechanisms.

**ii. Assumptions about the context of execution of the system under text**

Fault injection methods involve manipulating the context in which the system under test executes during normal operation such as changing code or the amount of supporting resources available. This means that assumptions about the context required for normal system execution are known which can be negated to identify system states, injection times or test points at which faults can be injected.

For example, consider an application requires 10 megabytes of memory to operate normally. Negating this assumption about memory context of the application would result fault injection tests being generated to test how the system performs when not enough memory is available.

## 7 NegTest Extension

In this section we introduce the design of an extension of a unit testing framework for negative testing called NegTest. NegTest is developed in Ruby programming language as an extension to the MiniTest unit testing framework which is the standard testing framework included with the ruby system. The extension is developed based on the Negation Testing perspective described in section 6 Figure 3 shows the basic structure of the main components of the extension.

The NegTest functionality is imported into MiniTest where a negation test case is defined. The test case specifies the assumptions about the inputs of the SUT and the expected behavior of SUT when tested with negated and non negated data.

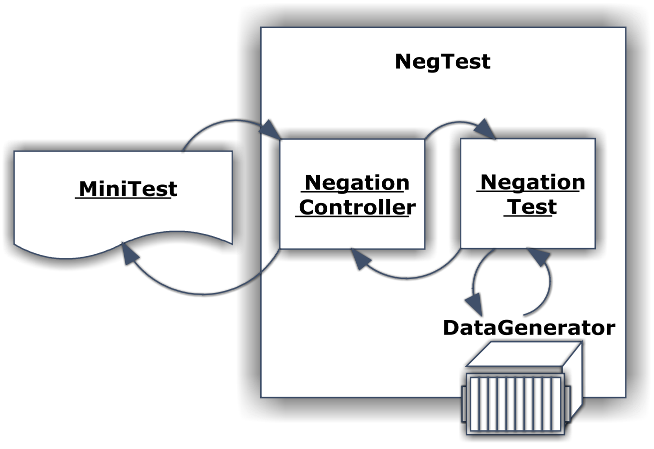


Figure 4 Main components of the NegTest extension

The Negation Controller module receives the negation test case which first builds a truth table of possible combinations of negated and non negated data depending on the number of assumptions of inputs specified. The NegationController randomly selects a combination from the truth table. For each assumption, a value of True causes the NegationTest module to randomly select amongst sub universes returning negated data from the data generator. A value of False causes the NegationTest module to return non negated data from the data generator. The returned sets of data are then tested against the SUT while checking that the expected behavior for negated or non negated behavior data is maintained. The results are then returned to MiniTest.

The sections that follow describe the main components of the NegTest extension.

### 7.1 Ruby

The extension is developed in Ruby programming language. Ruby is a high level dynamic language which means that it allows a program to modify itself at runtime. This makes it easy to extend the programs with new functionality. It is also an object oriented language having all its features implemented as objects. A key feature of Ruby is duck typing [14] which means that objects are described by specific data types, but rather by what the objects can and cannot do. This is particularly useful when testing as it allows any data to be run against the SUT to reveal faults. Another key feature of Ruby is the possibility to pass code segments called blocks from one object to another. This is important when defining templates of test cases as will be shown in the design of the extension. These features make ruby an attractive programming language to use for this study.

### 7.2 Unit Testing Framework – MiniTest

MiniTest is the standard unit testing framework included with the Ruby system version 1.9. It is an updated version of its slower predecessor called Test Unit. MiniTest provides a small and fast unit testing framework with a vast number of assertions that can be easy extended with new functionality. The test cases defined within MiniTest are similar to those defined in other testing framework which is an advantage as the time required learning how to use the tool is shortened. Consequently, an extension with similar structure will also be quick to learn MiniTest also provides features for Behavioral Driven Development (BDD) testing, a Bench marking feature which assesses the performance of algorithms and a mock object framework for stubbing out code.[14] This set of features makes MiniTest a good tool for the negative testing extension.

### 7.3 Data Generator – Rantly

The data generator used for the extension is extracted from Rantly testing tool. The generator has features for generating the following kinds of random data: fixed numbers, floats, strings, ranges of numbers or characters, strings generated from regular expressions, selecting values from a set, controlling the frequency of the kind of data generated and guards for filtering data. One major advantage of Rantly over other generators is it does not depend on any predefined data dictionaries for generating from which data is retrieved. Instead Rantly has functions that generate different forms of random data. Table 2 shows some ruby random data generators and identifies key differences between the generators features

| Generator | Dictionary / Pre defined values | Regular expression | Guards |
|---|---|---|---|
| Rantly | | X | X |
| Faker | X | | |
| Forgery | X | | |
| Randexp | X | X | |
| Random data | X | | |

**Table 2: Ruby Random Data Generators from www.rubygems.org**

### 7.4 NegTest Extension

The NegTest extension provides logic for manipulating user specified input assumptions into a form that can be used to get data from the data generator. The generated test data is then run against the end user specified oracles to test the SUT. The extension consists of two main modules, a NegationController and a NegationTest module. In this sub section, a simple example is introduced which is used to explain aspects of the design and is followed by a description of the NegationTest and NegationController modules

#### 7.4.1 A simple example

A simple program input_int_1_10 (input) takes a single string value as input. It must be possible to convert the input into an integer. The program verifies that the value is in the range 1-10. The method returns the value if it is in the range 1-10. Below is the pseudo code for the method:

```
Convert input into an integer
If 1 < = input < = 10
return input
Else
return "Must be an integer in the range [1, 10]"
```

A ruby tester might define the following test cases using MiniTest:

**Positive test cases**

```
 def test_int_1
   assert_equal 1, input_int_1_10(1)
  end

  def test_int_2
   assert_equal 2, input_int_1_10(2)
  end
```

**Negative test cases**

```
def test_int_0
   assert_equal "Must be an integer in the range [1,
10]", input_int_1_10(0)
  end

  def test_int_11
   assert_equal "Must be an integer in the range [1,
10]", input_int_1_10(11)
  end

def test_string_11
   assert_equal "Must be an integer in the range [1,
10]", input_int_1_10("11")
  end
```

The tests above are not exhaustive but are an example of the tests expected to be generated by the NegTest extension.

#### 7.4.2 Characteristics implemented in the design

NegTest is designed to test the SUT with different kinds of negated and non-negated inputs. Using the perspective of Negation Testing which is described in section 6, the extension provides features for defining negated and non – negated assumptions about the inputs. The extension randomly selects between the negated and non-negated assumptions to generate negated or non – negated data respectively.

As NegTest is an extension of a unit testing framework, it must exhibit the main characteristics of a unit testing framework. That is to say, it must provide features for the definition of a test case which includes an initialization of input values, the

SUT and specification of an oracle which returns the result of the test. In addition, in the event that a failure is encountered, the error details should be displayed and no other test should be run.

The NegTest extension is designed to verify that the SUT returns the expected response when tested with either negated or non negated inputs. A fault in the SUT is discovered if unexpected behavior is realized while testing with either negated or non negated data. A Non Negated Input Response Oracle (N-NIR0) and a Negated Input Response Oracle (NIRO) is defined.

The Non Negated Input Response Oracle (N-NIRO) returns a decision based on the expected behavior of the SUT for non - negated input. Following the simple example described in sub section 7.1, any input that is in the range 1 to 10 is non negated input and hence the Non Negated Input Oracle (N-NIRO) expects the response from the SUT to be the input supplied from the range 1 to 10. In such a case, the oracle will return a value of true. If the response from the SUT is not the same as the input supplied, the oracle returns false.

Table 3 illustrates the behavior of the N-NIRO described above. The table shows that when a non negated input of 2 is returned by the SUT, N-NIRO returns true. When the error message is returned for the negated input of 11, N-NIRO returns false. Table 3 also introduces test types A and C that represent tests of the SUT using the N-NIRO with non negated and negated data respectively. When the N-NIRO returns true for Test A, the test passes or is successful. Otherwise the test fails. When the N-NIRO returns false for Test C, the test passes or is successful. This is because we expect the SUT to not return the negated input, but return the error message.  Otherwise the test fails.

| **Test Type** | **Input Type** | **Example** | **SUT Response** | **N-NIRO Reponse** | **Test Status** |
|---|---|---|---|---|---|
| A | Non Negated | 2 | 2 | True | Pass |
| C | Negated | 11 | *Must be an integer in the range [1, 10]",* | False | Pass |
| A | Non Negated | 2 | null | False | Fail |
| C | Negated | 11 | *11* | True | Fail |

Table 3 Non Negated Input Response Oracle behavior

The Negated Input Response Oracle (NIRO) returns a decision based on the expected behavior of the SUT for negated input. Following the simple example described in sub section 7.1, any input that is not in the range 1 to 10 is negated input and hence the Negated Input Oracle (NIRO) expects the response from the SUT to be the error message. The oracle will return a value of true. If the response from the SUT is not the error message, the oracle returns false.

Table 4 illustrates the behavior of the NIRO described above. The table shows that when the error message is returned by the SUT because of the negated input of 11, NIRO returns true. When any value other than the error message is returned for the negated input of 11, NIRO returns false. Table 4 also introduces test types B and D that represent tests of the SUT using the NIRO with non negated and negated data respectively. When the NIRO returns false for Test B, the test passes or is successful. This is because we expect the SUT to not return the error message, but return the non negated input.  Otherwise the test fails.
When the NIRO returns true for Test D, the test passes or is successful. Otherwise the test fails.

| **Test Type** | **Input Type** | **Example** | **SUT Response** | **NIRO Reponse** | **Test Status** |
|---|---|---|---|---|---|
| B | Non Negated | 2 | 2 | False | Pass |
| D | Negated | 11 | *Must be an integer in the range [1, 10]",* | True | Pass |
| B | Non Negated | 2 | *Must be an integer in the range [1, 10]",* | True | Fail |
| D | Negated | 11 | *11* | False | Fail |

**Table 4 Negated Input Response Oracle behavior**

### 7.4.3 NegationTest module

The NegationTest module processes end user specified assumptions and returns negated or non – negated random data from the data generator. Fixnum, float, string, Boolean and nil types are the default sub universes provided by NegationTest module. If an assumption states that non negated data consists of Fixnums, the NegationTest module randomly selects a sub universe from floats, string, Boolean and nil types and a value is return from the selected sub universe. The NegationTest module provides classes and methods for getting fixnums, floats, string, Boolean and nil values from the data generator from which other custom sub universes can be constructed. The module defines the syntax used by the end user to specify the input assumptions. The module also allows custom generators to be defined by the end user for situations in which complex types of data are required. The test data generated is returned to the NegationController module.

### 7.4.4 NegationController module

The NegationController module receives user defined assumptions which are forwarded to the NegationTest module to obtain input data from the random generator.

When the NegationController receives input assumptions, a truth table is constructed. The different combinations of True and False values are then used to direct the NegationTest module to return negated or Non Negated data. A True value returns negated data and a false value returns non negated data.

The NegationController has the logic that will run the generated inputs against the N-NIRO and NIRO to test the SUT and report the results to MiniTest.

## 7.5 Techniques implemented in the design

The test techniques implemented in the design include equivalence partitioning, exploring all allowable character sets and data types and find input values that may interact and test combinations of their values whose descriptions can be found in sections 5.9, 5.3 and 5.4 respectively.

The inputs generated by the extension are either in the negated or non negated classes depending on the assumption specified by the tester. This functionality is distributed across strings, numbers, Boolean and nil values which are the basic types in ruby that can be used to construct complex types such as arrays in Ruby. In order to test different combinations of inputs, the NegationController creates a truth table for each input or set of input assumptions and which is used to direct the NegationTest module to create negated or non negated data.

## 8 Evaluation and Results

The evaluation was carried so as to ascertain that the NegTest tool can be used to discover faults in applications. Various test candidates were used in the evaluation and are listed in Table 5. For each candidate, select methods were identified that take one or more parameters. Each candidate was tested 10 times so as to ensure that different types of data and different combinations of data for multiple parameter methods can be tested. NegTest will run a total of 200 tests assuming no failures are encountered to stop execution. Therefore a total of 2000 tests were run for each test candidate.

| **Test Candidate** | **Methods** | **#Parameters or type** |
|---|---|---|
| Bookland | EAN.valid<br>ISBN.valid<br>ISBN10.valid<br>ISBN.to_isbn_10<br>Identifier.checksum<br>Identifier.payload | 1<br>1<br>1<br>1<br>1<br>1 |
| Chronic | parse | 1 |
| Versionomy | create | 5 |
| ICalendar | add_event | event object |
| Simple Statistics | mean | 1 array on numbers |

Table 5: Test Candidates from www.rubygems.org

Table 6 shows the results obtained from testing the test candidates with the NegTest extension. The table shows the average time (Avg Time) taken for the 10 test runs of each test candidate with each run consisting of a maximum of 2000 possible tests. The table shows the total number of failures for the four test types A, B, C and D described in section 7.4.2 Table 3 and Table 4, the total number of successful tests run before a fault or unexpected behavior is encountered and the total number of tests not run.

## 9 Discussion

The results of the study show that using the Negation Testing approach, test techniques can be automated so as to be able to generate test cases that can discover unexpected or unexplored behavior in real systems. NegTest generated test cases consisting of negated inputs that brought about failures, many of which were exceptions that were caused by inputs of the wrong data type. In addition, NegTest was able to identify a fault in one test candidate that did not cause the test candidate to fail. The Chronic test candidate expects unique string key words which are parsed and the corresponding date returned. However negated random string input combinations were able to return a valid date and hence an indication of a fault is the program.

The study shows that Negative Testing is not clearly defined. The study described four different definitions of Negative Testing with each definition seeking to test one or more aspects about systems such as inputs and execution context. The definitions described were not exhaustive, but their existence indicates the need for a general and clear description of Negative Testing.

In the study test techniques that can be categorized as negative testing techniques were described in terms of their purpose, and strengths and weaknesses of the techniques with accompanying examples. Analysis of the techniques revealed gaps in the current understanding of negative testing. The gaps occur because no clear description of negative testing exists. In Section 2, four different definitions of NT were introduced that test different aspects about systems. This means that the NT techniques can be executed in four different ways which leaves testers with the challenge of selecting definition to follow. Therefore a more general definition is required that clearly describes NT.

| **Candidate Method** | **Avg Time (secs)** | **A** | **B** | **C** | **D** | **SuccesfulTests Run** | **Tests Not Run** |
|---|---|---|---|---|---|---|---|
| EAN.valid | 0.014001 | 0 | 0 | 10 | 0 | 1014 | 986 |
| ISBN.valid | 0.013301 | 0 | 0 | 10 | 0 | 1024 | 976 |
| ISBN10.valid | 0.013401 | 0 | 0 | 10 | 0 | 1008 | 992 |
| ISBN.to_isbn_10 | 0.019601 | 0 | 0 | 10 | 0 | 988 | 1012 |
| Chronic.parse | 0.403323 | 0 | 0 | 0 | 2 | 1912 | 88 |
| Versionomy.create | 0.158009 | 0 | 0 | 10 | 0 | 74 | 1926 |
| ICalendar | 0.029001 | 0 | 0 | 10 | 0 | 1001 | 999 |
| Simple Statistics | 0.043002 | 0 | 0 | 0 | 10 | 1034 | 966 |

**Table 6: Test Results**

Furthermore, the definitions state aspects such as inputs and context of execution which are to be explored when testing the SUT with test techniques. But the definitions do not state how these aspects are derived or used to derive negative test cases.
In addition, some techniques that can be used for negative testing can also be used for positive testing raising the question of whether negative testing is a technique or an approach that can be applied to any test technique.

Negation Testing described in section 6 fills the gaps mentioned above describing negative testing as an approach that can be applied to any test technique and also specifies how aspects of testing can be derived from system descriptions and used for negation testing

The design of the NegTest extension described in section 7 shows that negative testing techniques can be automated using the negation testing approach. Three techniques described in section 7.5 are implemented in the design. These techniques were selected because they explore the inputs aspect of the SUT which are derived from the SUT specification, and do not require knowledge of various structures used in the SUT.

The NegTest extension was designed with the Ruby system which comes with a standard unit testing framework called MiniTest. MiniTest is a fast testing tool which can be seen from the short time taken to execute the tests in Table 6.

Ruby was particularly useful because of its duck typing feature that allows objects to be described by what they do rather than being associated with a specific data type. This feature makes it possible to assemble different types of data in a single data set which can be used to test the SUT.

In addition, Ruby has methods that enable permutations and combinations of data to be computed. These methods were considered when designing the NegTest extension.

Using the design described in section 7, it was possible to define input assumptions of various complexities for the test candidates using both the default and custom input generators. However this was challenging for test candidates in which some but not all the characteristics of the input assumptions was known. In such a case, it was difficult to determine if certain unexpected behavior.

The NegTest extension did not perform in terms of running a complete set of tests. This is because when a failure was encountered, no other tests could be run. This is the default behavior of a unit testing framework. Because of this the remaining test data is never used to test the SUT for unexpected behavior. This is shown in table 6 as the tests not run against the candidates. It is desirable to have all test data tested. As further work, the design should allow all tests to be run.

The NegTest extension explored the test candidates using the four kinds of tests A, B, C and D. Some of the failures realized by the NegTest extension did not bring about failures in the candidates. Instead the candidates showed new behavior that would need to be explored so as to ascertain the conditions under which they occurred. This shows that Negative testing should not only focus on finding those use cases that cause the SUT to fail, but to also cause the SUT to exhibit new behavior that does not result in failure of the SUT.

The results in Table 6 show that in some cases more negated inputs were generated than the non – negated inputs and vice versa. This is because the generation of negated or non –negated inputs is random. As further work, fairness could be implemented so that an equal number of non – negated and negated inputs to be generated.

In addition, the design could be extended to incorporate other test design techniques using the different aspects of Negative Testing identified in section 2. The current design uses input assumptions to test the SUT. Further work could explore different ways of testing the context of execution of the SUT.

For the evaluation of NegTest, there were 10 test runs for each candidate totaling to 2000 tests. The aim was to be able to test the candidates with as much random data as possible. Key questions on Negation Testing would be on how many tests should be run, how many sub – universes should be used for negation and generally the time used for testing. One recommendation would be to use the assumptions indentified as a control and test each assumption. The assumptions about the SUT should be tested within the allowed time and budget for the current release. Other assumptions are carried on to the next release and tested then. This is not a full proof solution. However the questions raised above can also be explored as further work.

## 10 Conclusion

Negative Testing is not a technique, but rather an approach to testing that can be applied to test techniques. It seeks to discover unexpected behavior of systems that may or may not result in the failure of the SUT. Negative Testing explores various aspects about systems and both negated and non-negated characteristics of these aspects should be tested. The automation of testing techniques to which the negative testing approach has been applied can reveal unexpected behavior from the SUT.

## 11 References

1. K. Naik, P.Tripathy. *Software Testing and Quality Assurance Theory and Practice*. New Jersey, John Wiley and Sons, 2008.
2. S. Eldh. “On Test Design.” PHD, Marladarlen University, Sweden, 2011.
3. “ Negative Test”. Internet: http://en.wikipedia.org/wiki/Negative_test [Jan. 13, 2012].
4. B. Beizer. Software Testing Techniques, VNR, International Thomson Computer Press, 2nd ed. Boston, 1990.
5. G. J. Myers. *The Art of Software Testing*. New Jersey, John Wiley And Sons, 2004

6. J. A. Whittaker. "*How to Break Software: A Practical Guide to Testing*", Addison-Wesley, 2003
7. V. Vaishnavi, B. Kuechler. "Design Science Research in Information Systems". Internet: http://desrist.org/design-research-in-information-systems/ [Jan. 13, 2012]
8. J.Lyndsay. Positive View On Testing.[On-Line] Available: http://www.workroom-productions.com/papers/PVoNT_paper.pdf [Jan 24,2012].
9. A.Cyrille, B.Armin, and H.Shinichi. Exhaustive Testing of Exception Handlers with Enforcer. [On-line] Available: staff.aist.go.jp/c.artho/papers/fmco-2006.pdf [Jan 24, 2012].
10. H. Haissam , R. Ayoubi , R. Velazco. A Survey on Fault Injection Techniques. [On-line] Available: www.citemaster.net/getdoc/9045/04-Hissam.pdf [Feb 20, 2012].
11. Y. Jia, M. Harman. An Analysis and Survey of the Development of Mutation Testing. [On-line] Available: http://www.dcs.kcl.ac.uk/pg/jiayue/repository/TR-09-06.pdf [Feb 18, 2012].
12. P. Amman, J. Offutt. *Introduction to Software Testing. Cambridge*, Cambridge University Press, 2008.
13. J. Arlat, A. Costes, Y. Crouzet, J. C. Laprie, D. Powell. "Fault Injection and Dependability Evaluation of Fault Tolerant Systems", IEEE Transactions on Computers, 42 (8), pp. 913-923, August 1993.
14. D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby. The Pragmatic Programmer's Guide*. Pragmatic Programmers, Addison Wesley , 2009.

# APPENDIX A –Negative Testing Techniques

Appendix A gives a detailed description of the techniques categorized as Negative Testing Techniques.

## 1. Fault Injection

This is a technique in which the quality of exception or dependability features of the SUT are tested and assessed by introducing faults or failures into the SUT [9]. The technique tests the possible configurations of the SUT and verifies that the SUT behaves as is stated in the specifications in the presence of faults.

When applied to software, this technique involves altering the state of the SUT by applying changes to the software that will result in the execution of exception or error handling code paths. The changes are applied to the SUT directly or a layer between the SUT and any supporting software such as an operating system from which faults are injected. This technique is effective in exploring fault and exception handling code that is often left untested by positive testing. The changes can take various forms such as code modification, erroneous input flags and memory faults

To execute the technique, the tester must have a good understanding of the operations, code and behavior of the system so as to be able to identify system states, injection times or test points at which faults can be injected. This requires an initial and in depth analysis of the SUT. The inputs for the technique include the specification, code, knowledge about the environment of the system and experience using the application [10].

There are two main ways in which the technique is applied to software i.e. Compile time injection and Run time injection.

Compile time injection involves modification of the source or assembly code to introduce faults into the SUT. The code is modified in areas in which there is sufficient interaction or communication between components for the fault to be realized. When the SUT is run, the faulty code is executed. As an example, consider a function that divides numbers as illustrated below:

```
 A=4/2;
```

A compile time injection example would involve modifying the source code to:

```
A=4/0;
```

This simple change is similar to the errors made by developers. Error handling mechanisms should detect such errors in software. Compile time injection does not allow faults to be introduced into the software as it is executing.

Runtime injection involves the use of a trigger to introduce a fault into running software. The trigger used to introduce the fault is implemented in three main ways:

i. Code insertion involves the addition fault injection code to the original source code that will inject the fault as the SUT is running. The key difference between code insertion and compile time injection is that the former involves addition of fault injection code which is executed at runtime whereas the latter modifies the original source code and the fault introduced at compile time

ii. Exception or trap transfers control to a fault injector when certain conditions are satisfied by the SUT. The fault injector introduces the fault and the SUT resumes execution with the new faulty state.

iii. Timeout involves the injection of a fault as the SUT is executing, but after a given amount of time. This method is suitable for simulating faults that occur periodically. The timer may be implemented in software or hardware and requires no modification of the source code.

For the technique to be effective for NT, faults introduced must cause code paths that handle faults and exceptions to be executed. In addition, the software used to inject faults into the system must not affect the system behavior i.e. the software is independent of the system under test and any errors within the fault injection software does not affect the system under test.

The techniques key strength lies in the fact that it targets fault and exception handling routines that are rarely tested using positive testing. The technique also enables applications and software to be tested which could not be realized with hardware fault injection. No special hardware is required to carry out the testing using the technique which reduces the cost of the testing effort.

However some of the methods used to execute the technique require the modification of the source code. This means that the software executed during the test is not the same as the software run by the end users and hence is not an accurate replica of the user context. In addition, errors within the introduced fault code can lead to incorrect test results about the system under test. Furthermore, areas of the SUT that cannot be accessed by software cannot be tested.

### 2. Mutation Testing

Mutation Testing is a fault-based testing technique in which faults are introduced into software by means of syntax changes. The modified software is then executed against a test set to verify that the faults are detected. The technique helps testers develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

The code containing the modified syntax changes is called a mutant of the original program. The technique only targets faults which are close to the correct version of the program with the hope that these will be sufficient to simulate all faults [11].

To execute the technique, the tester requires knowledge of the different parts of the source code that can be modified to test the system for faults. The tester modifies the original program, adding small syntactic changes to create mutants or the modified program. For example, given the Boolean expression below:

```
If (A <B) return 1;
```

A possible mutant could be:

If (A>B) return 1;

Any statement within software can be mutated as the statements shown in the statement above and thus the number of mutants that can be created from software depends on the size of the software.

For large software, mutation testing becomes impractical due to the high computational cost of executing the enormous number of mutants against a test set. To tackle this problem, techniques that attempt to reduce the number of techniques and increase speed of execution are being developed. In [11] the following cost reduction techniques are identified:

- Mutant sampling: This randomly selects a set of mutants from a group of mutants.
- Mutant clustering: utilizes clustering algorithms to identify mutants.
- Selective Mutation: Seeks to identify a set of all mutants that can be derived from mutation operators without reducing test quality.
- Higher order mutation: seeks to identify uncommon mutants that detect unique faults.

Another set of cost reduction techniques focuses on optimizing the process by which mutants are run [11]. These include Strong, Weak and Firm Mutation and run time optimization.

Like fault injection in the previous section, mutation testing requires the source code to be modified meaning that the conditions under which the software is executed during the test are not the same as when the software is executed by the end user and hence is not an accurate replica of the user context.

## 3. Apply inputs that force all error messages to occur

This technique focuses on the utilization of input values that will cause the SUT to exhibit error messages with the aim of ascertaining that the errors or behavior produced matches the behavior stated in the system specification. This technique targets the error handling code which is often difficult to develop for various situations.

In [6], the following aspects of input data that are targeted:

i. Invalid data types will generate errors for example, providing an integer where a string in expected
ii. Providing inputs or sets of inputs whose length is greater than the expected length for an input, or providing a null value.
iii. Considering the boundary values of inputs which tend to reveal errors as these values are not handled well by developers.

To execute this technique, information about the characteristics of the inputs is required, in particular the data type, length of the inputs and boundary information. These can be obtained from the specification of the software or the source code. In addition, knowledge about the errors messages and the conditions under which the errors are invoked is required.

For example, given an input field that accepts only characters, entry of a numeric value should generate an error in the application.

The technique is easy to use for software that is well specified to include the error messages and conditions as the tester simply refers to the specification to derive the test. The technique can help reveal input data that can bring about subtle, rare or unique errors that the developers did not anticipate [6].

However the technique is limited by the fact that it doesn't account for situations in which input validation that results in the generation of error messages occurs at another layer of abstraction [2]. In this case subjecting a component to a series of tests will result in numerous failures. Therefore, the tester must fully understand the implementation of the application to effectively use this technique.

**4. Explore allowable character sets and data types**

This technique checks whether the SUT has validation mechanisms to prevent failures caused by character sets or data types that the SUT was not designed to use.

Different operating systems and programming languages based on either Ascii of Unicode character sets have reserved control structures or symbols that the SUT must be able to handle. For example the C language has the ++ symbol for incrementing numerical values. The SUT must be able to validate such inputs and return the appropriate error responses.
To execute this technique, knowledge about the platform on which the application is to be developed is required. This information can often be obtained from the specification document. In addition, the SUT may be tested with various data types offered by the programming language in which case the SUT must be able to validate the invalid data types.

It should be noted that the test should not be limited only to the application under test but also the external systems the application interacts with which could supply erroneous values.

The technique is rather complex to execute. This can be seen from the reference table provided in [6] (Table 2.2 on page 29-33) to guide testers in executing the technique.

Table 2.2 on page 29-33 in [6] provides a guide for testers to use in executing the technique. The technique is efficient in revealing validation errors. However, numerous errors will be found if validation is done at a higher level of abstraction.

**5. Find input values that may interact and test combinations of their values**

This technique involves testing the SUT with combinations of inputs that are related or are involved in the same operations. The goal is to identify inputs that affect each other in computations or utilize the same resources and test different combinations of these inputs so as to ascertain that the SUT functions for all possible input combinations.

For example, for a function that adds two of its parameters, the parameters are the values whose combinations are to be tested.

[6] suggests that the values in a relationship should describe aspects of a common internal resource or be used in the same internal computation or calculation. This information can be derived from the design of the application and the source code.

The major drawback of the technique is that it is not possible to test all input combinations. Whittaker suggests selecting a good subset from the possible combinations. A possible solution would be selecting a representative value from different partitions identified using the equivalence partition method.

**6. Repeat the same input or series of inputs numerous times**

The technique tests whether an application has error handling procedures for situations in which the context of execution is modified as a result of all resources that the SUT is dependent on being consumed [6].

Continuous repetition of the same inputs uses up system resources such as memory buffers, data storage space or communication with remote resources. In addition, if an application uses a remote resource, it is difficult for the application under test to know of the remote systems limitations and hence developers must have proper mechanism to prevent the system under test from crashing.

To execute the technique, the tester needs a good understanding of the application inputs, operation and the environment or external resources that the application interacts with. The inputs that the user is expected to use are good candidates for the test. Then the way the inputs are used internally is considered i.e. if they use up resources or interact with other systems. These inputs are then used to execute the same procedure until the application fails.

The technique is good for stress testing the application under test and can also benefit from automation as it involves executing the same task repeatedly [2].

**7. Force a data structure to store too many or too few values**

This technique checks whether appropriate controls and error handling procedures on data structures or resources are in place so as to avoid underflow, over flow or corruption of data [6]. The test is executed by providing too many, too few or invalid inputs to the SUT.

The technique reveals errors brought about by the developers neglect of limitations of the different data structures they use. The data structures also include external resources that the application uses.

To execute the technique, the tester requires knowledge about the different data structures and resources that can be exploited of the application. The source code and information about the applications environment are good sources of information. An example of a simple data structure is an array whose size is not checked before attempting to add a new value to the array. Addition of a value when the array has reached its capacity will result in an error.

### 8. Force the media to be busy or unavailable

This technique tests the SUT to verify that it operates appropriately during error conditions related to media. The technique verifies that the applications return the correct error codes associated with different kinds of media problems.

This is common in concurrent applications that try to access a single resource. For example, a multithreaded application that updates a database will cause a deadlock when more than one process attempts to update the database.

To execute the technique, the tester requires in depth knowledge about the operation and environment of the system under test.

The drawback with this technique is that the test scenarios are difficult to simulate often requiring more resources than is available to testers. In addition, some applications have concurrent processes and it may be difficult to simulate a scenario in which different processes access the same resource at the same time.

### 9. Equivalence Partitioning

This technique divides software inputs into partitions from which test cases can be derived to uncover classes of errors. A single value from a partition will be treated similarly by the component producing the same system behavior and thus can be used to represent the entire class [12].

This technique can be used for both positive and negative testing. When used for positive testing, the focus is on those classes of inputs that cause the system to exhibit the expected system behavior. When used for negative testing, the focus is on those classes of inputs that cause the system to exhibit unexpected system behavior.

Partitions of inputs are identified from the characteristics of the program, inputs, environment and behavior of a given software unit. The characteristics can be derived from the specification and constraints of the software. The inputs can be method parameters, global variables, objects representing current state or user level inputs to a program, depending on the kind of software artifact being analyzed [12].

[12] suggests the following general strategies for identifying the different partitions from each characteristic or constraint:

- Valid values that cause the system to exhibit expected system behavior should be considered. This also applies to ranges of values which can be divided into smaller sets of partitions exercising different functionality. This is because different combinations of the inputs can result in unexpected system behavior and thus must be explored
- Invalid values that cause the system to exhibit unexpected system behavior should be considered. This should not be limited to the details stated in the specification, but should include less obvious values such as wrong data types or null values. One way to identify the less obvious values is to obtain values that contradict the characteristics or constraints of the inputs [8]. For example, use of characters where integers are required.
- Boundary values should be considered i.e. values that are at or close to the boundaries of the input. For example, given a range
- Check that no value belongs to more than partition so as to avoid duplication in the results

A key strength of the technique is that there is little training required and the tester does not need to understand the implementation. This is due to the fact that the classes are derived from the specification of the application that contains characteristics usually referring to a single parameter.

However, it is often the case that specifications are incomplete and therefore not all classes can be explored. Furthermore, in some situations the tester will only have access to the software's executable and general knowledge of the functions of software, but not the specification. In this case, identification of the partitions is more of guess work.

This technique has high chances of uncovering unknown system behavior as it generates test inputs or scenarios that are outside what is stated in the specification. However, for well specified systems such as safety critical systems, less unknown system behavior may be discovered.

## 10. Boundary value analysis

This technique examines the boundary values of the characteristics or properties of software as more errors tend to occur at the boundaries. The technique is often considered an extension of equivalence partitioning in which the boundaries values of the partitions are the major areas of interest.

[5] identifies the following differences between boundary value analysis and equivalence partitioning:

- Rather than selecting any element in an equivalence class as being representative, boundary-value analysis requires that one or more elements be selected such that each edge of the equivalence class is the subject of a test.
- Rather than just focusing attention on the input conditions (input space), test cases are also derived by considering the result space (output equivalence classes).

The technique requires that the lower and upper boundaries of a property or characteristic be examined. For each boundary, a value at the boundary, one value below the boundary and one value above the boundary is required. The technique can be used for both positive and negative testing. When used for negative testing, values below or above the boundary is of interest.

Partitions of inputs and outputs are identified from the characteristics of the program, inputs, environment and behavior of a given software unit. The characteristics can be derived from the specification and constraints of the software. The inputs can be method parameters, global variables, objects representing current state or user level inputs to a program, depending on the kind of software artifact being analyzed [12].

Boundary Value Analysis is valuable technique that can reveal errors if used correctly. It is easy to execute, the major difficulty being the identification of the boundary conditions. If used correctly the technique can discover boundaries overlooked by programmers, discover boundaries resulting from interactions amongst subsystems and validate requirements.

# APPENDIX B – NegTest Design

Appendix B gives a description of the design of the NegTest extension

## Introduction

NegTest is designed as an extension to the MiniTest unit testing framework with the purpose of automatically generating test cases and testing the SUT. Section 7.4.2 of the paper described characteristics implemented in the design which can be interpreted as the general requirements to be achieved by the design of the extension.

The diagram below shows the major components of the NegTest extension.

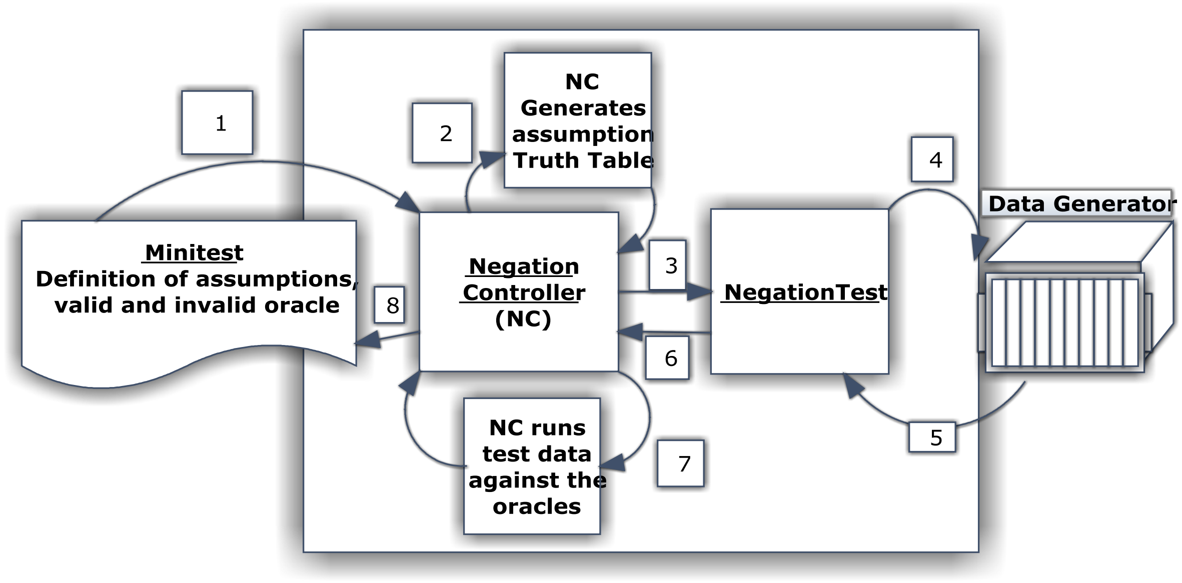

1
2
NC Generates assumption Truth Table
4
Data Generator
Minitest
Definition of assumptions, valid and invalid oracle
8
Negation Controller (NC)
3
NegationTest
6
5
NC runs test data against the oracles
7

**Figure 5: Flow of Control in NegTest Components**

Below is a description of the flow of control in the NegTest extension shown above.

1. The end user defines the input assumptions, the valid and invalid oracle and submits the test.
2. The NegationController receives the assumptions and generates a truth table based on the submitted assumptions.
3. The NegationController randomly selects a set from the truth table and selects a single value that is used to set a variable that informs the NegationTest module if negated or non – negated data should be returned. The assumption is forwarded to the NegationTest module
4. Based on the assumption and the status variable, the Negation test module queries the data generator
5. The data is returned to the NegationTest module
6. The test data is returned to the NegationController module
7. The NegationController runs the generated inputs against the Non –Negated Input Response Oracle and the Negated Input Response Oracle
8. The NegationController reports the results to MiniTest.

Three techniques are implemented in the design. In the design, the techniques are dependent on one another as described below:

**a) Equivalence Partitioning**

The NegTest extension generates two classes of input which are negated and non – negated data. This is achieved by selecting a Boolean value from a truth table generated from the assumptions submitted for testing. If a value of True is selected, negated data is returned. Otherwise non – negated data is returned.

**b) Find input values that may interact and test combinations of their values**

For tests that involve multiple input assumptions, a truth table is constructed giving the different combination of True and False value pairs for the inputs. When generating a single set of values for the assumptions, a single set is selected from the truth table and for each assumption the corresponding True or False value is selected to generate data as is described above for equivalence partitioning.

**c) Explore allowable character sets and data types**

For each type of data to be returned, NegTest requires a specification of the non – negated form of the input. The negated form is them any other type that is not equal to the non – negated type. For example is an assumption requires integers as non negated data, the negated data would consist of strings, floats, Boolean values etc.

The design requires a negated and non – negated routine to be developed. In the negated routine, the other types not equal to the non – negated type are randomly selected and the data generator returns this data.

The selection between the non – negated and negated routine is based on the Boolean value select from the truth table

The sections that follow describe the design of the NegationTest and NegationController in detail.

## 1 NegationTest

The NegationTest module defines accessor methods and assumption classes that are used to query random data from the random data generator. This section describes the components that enable the NegationTest module to perform its task.

### 1.1 Assumption Classes

The assumption classes define methods that interact with the data generator to return specific kinds of data. The classes use data generator public methods and syntax to get data. All classes inherit from the Assumption class which defines a single public method that "generate" that is used to query the data generator for data.

The child classes use the ruby construct called a Proc to define the kind of data that is required. A Proc is a block or set of ruby statements that can be bound to a variable. The Proc defined in a child class is then passed to the parent Assumption class which will query the data using the generate method.

Below is class hierarchy of the assumption classes.

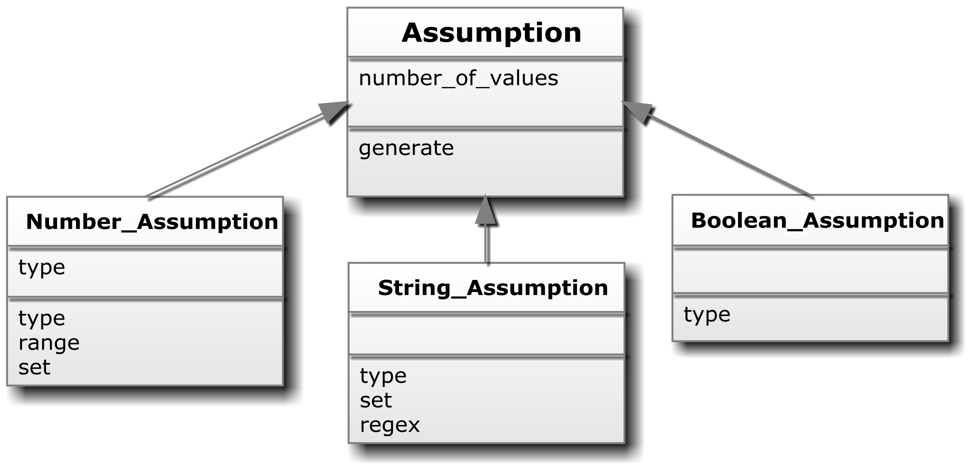


Figure 6: Assumption class hierarchy

Three main assumption child classes are provided which are used to query the main types of data in ruby which include Fixnum (Integers), Floats, String, Boolean and nil. Each class specifies different ways by which these types can be specified.

All methods in the child classes receive a single Boolean variable which is used to determine if negated data should be returned (True), and if non – negated data should be returned (False). This means that each method defines how negated and non – negated data will be queried from the data generator.

The module defines a module parameter is_violated which is used to determine if negated (is_violated=True) or non - negated (is_violated=False) data should be returned

Each of the classes defines various methods. Each method must define a routine that is used to return negated or non-negated data.

The sub sections that follow provide a description of the classes.

### 1.1.1 Number_Assumption class

This class queries the data generator for negated and non- negated fixnum and float values. It has the following methods:

i. Type

This method queries the data generator for fixnum and float values when non – negated values are required. The method will return any fixnum in the range -1073741823 to 1073741823. The method will return any float in the range Float::MIN to Float::MAX. Float::MIN and Float::MAX are special ruby constants

The method takes a Boolean value as a parameter that is used to determine if negated or non negated data should be returned. It also takes a fixnum value that indicates how many numbers should be returned.

When negated fixnum values are required, the random generator is queried to randomly select a single data type from a set consisting of a float, Boolean, string or nil value. Based on the selected data type, a corresponding value is returned.

Similarly, when negated float values are required, the random generator is queried to randomly select a single data type from a set consisting of a fixnum, Boolean, string or nil value. Based on the selected data type, a corresponding value is returned.

ii. Range

This method queries the data generator for fixnum of floats in a specified range. The method takes a Boolean value as a parameter that is used to determine if negated or non negated data should be returned. It also takes a fixnum value that indicates how many numbers should be returned.

In addition, the method takes an array of range objects as a parameter. The range objects consist of the upper and lower limit values of a given range. Multiple ranges can be specified.

When non – negated data is required, fixnums or floats that are in the ranges specified are returned.

When negated fixnums are required, either fixnums not in any of the ranges are returned or the random generator randomly selects a single data type from a set consisting of a float, Boolean, string or nil value. Based on the selected data type, a corresponding value is returned.

Similarly, when negated floats are required, either floats not in any of the ranges are returned or the random generator randomly selects a single data type from a set consisting of a fixnum, Boolean, string or nil value. Based on the selected data type, a corresponding value is returned.

iii. Set

This method receives a set of fixnum or float values which are considered to be the only non – negated values. The values in the set must be returned whenever non – negated data is required.

The method takes a Boolean value as a parameter that is used to determine if negated or non negated data should be returned. It also takes a fixnum value that indicates how many numbers should be returned.

When negated fixnums are required, any fixnum not in the set and any value of float, string Boolean or nil type can be returned

When negated floats are required, any float not in the set and any value of fixnum, string Boolean or nil type can be returned

### 1.1.2 String_Assumption class

This class queries the data generator for negated and non- negated string values. It has the following methods:

i. Type

This method queries the data generator for string when non – negated values are required. The method takes a Boolean value as a parameter that is used to determine if negated or non negated data should be returned. It also takes a fixnum value that indicates how many numbers should be returned.

When non – negated data is required, random string values are returned. When negated data is required, any value of a fixnum, float, Boolean or nil data type is returned.

ii. Regex

This method queries the data generator for string values based on simple regular expressions. The method takes a Boolean value as a parameter that is used to determine if negated or non negated data should be returned. It also takes a fixnum value that indicates how many numbers should be returned.

When non – negated data is required, random string values are returned. When negated data is required, any value of a fixnum, float, Boolean or nil data type is returned or a string that does not match the regular expression is returned.

iii. Set

This method receives a set of string values which are considered to be the only non – negated values. The values in the set must be returned whenever non – negated data is required.

The method takes a Boolean value as a parameter that is used to determine if negated or non negated data should be returned. It also takes a fixnum value that indicates how many numbers should be returned.

When negated strings are required, any string not in the set and any value of float, fixnum Boolean or nil type can be returned

### 1.1.3 Boolean_Assumption class

This class queries the data generator for negated and non- negated boolean values.  It has the following methods:

i. Type

This method queries the data generator for Boolean values when non – negated values are required. The method takes a Boolean value as a parameter that is used to determine if negated or non negated data should be returned. It also takes a fixnum value that indicates how many numbers should be returned.

When non – negated data is required, random boolean values are returned. When negated data is required, any value of a fixnum, float, string or nil data type is returned.

## 1.2 Accessor Methods.

The Accessor methods provide a means through which the assumption classes can be used to get data. They form the syntax used by the end user to specify assumptions of inputs. Some of the methods can only used as a parameter for another accessor when stating a specific characteristic about an input assumption. Below is a description of the methods

i. Integer

This method takes another accessor method as a parameter and returns non- negated or negated fixnum values based on the parameter. If no parameter value is provided, the type method of the Number_Assumption class is used to query data.

ii. Float

This method takes another accessor method as a parameter and returns non- negated or negated float values based on the parameter. If no parameter value is provided, the type method of the Number_Assumption class is used to query data.

iii. String

This method takes another accessor method as a parameter and returns non- negated or negated string values based on the parameter. If no parameter value is provided, the type method of the String_Assumption class is used to query data.

iv. Boolean

This method returns non- negated or negated boolean values. The type method of the Boolean_Assumption class is used to query data.

v. Range

This method takes an array of ranges as a parameter and returns an array of range objects. This method must be used as a parameter.

vi. Set

This method takes a list of values as a parameter and returns the list. This method must be used as a parameter.

vii. Regex

This method takes a regular expression as a parameter and returns the expression. This method must be used as a parameter for the String method.

### 1.3 Default Syntax

Below is the allowed syntax provided by the module by default.

Integer () – returns non – negated or negated fixnums
Integer(range([RangeA,RangeB])) – returns non – negated or negated fixnums in the range RangeA and RangeB. RangeA and RangeB are expressed in the form a..z where a and z are fixnums

integer(set(a,b,c)) – returns non – negated and negated fixnums using the set a, b and c. a, b and c are fixnums

float() – returns non – negated or negated floats float(range([RangeA,RangeB])) – returns non – negated or negated floats in the range RangeA and RangeB. RangeA and RangeB are expressed in the form a..z where a and z are floats

float(set(a,b,c)) – returns non – negated and negated floats using the set a, b and c. a, b and c are floats

string() – returns non – negated or negated string values

string(set(a,b,c)) – returns non – negated and negated strings using the set a, b and c. a, b and c are strings

string(regex(expression) – returns non – negated and negated strings based on the regular expression.

### 1.4 NegationController module.

The NegationController module provides a single Controller class with methods for receiving user specified assumptions, oracles and running the tests against the SUT. This section describes the components that enable the NegationController module to perform its task.

#### 1.4.1 Controller class

The Controller class receives user assumptions. The assumptions are defined using the accessor syntax defined in the NegationTest module. The accessors are added to the Controller class using ruby mixin syntax (Include NegationTest). By using the mixin syntax, the accessors can be accessed within the Controller class as instance methods of the Controller class.

The Controller class defines the following methods:

i. Initialize

This is method receives user specified assumptions as ruby Procs and the user defined oracles as blocks. Both Procs and blocks consist of ruby statements. The method sets the number of tests to be run with a value of 100.

ii. Get_permutation.

This method receives a list of assumptions and creates a truth table of all possible negated and non - negated combinations of the input variables. For example, if the assumption represents a single value, the table will consist of only a single True and False value. If assumptions for two inputs are provided, the following combinations are provided

| Single Input | | Two Inputs | |
|---|---|---|---|
| Input 1 | | Input 1 | Input 2 |
| True | | True | True |
| False | | True | False |
| | | False | True |
| | | False | False |

Table 7: Assumption Truth Table

iii. Single_assumption

This method is executed when a single assumption is provided. It returns a value based on the assumption and the randomly selected Boolean value from the truth table which determines if a negated or non – negated value should be returned.

iv. Assumptions

This method is executed when multiple assumptions are provided. It returns a set of values based on the assumptions and the randomly selected set of Boolean values from the truth table which determines if a negated or non – negated value should be returned. For example if a 2 input values are provided, a truth table similar to the one in table 6 is created for the two inputs. The assumption method will randomly select a single row from the table and for each input, set the is_violated variable with the appropriate Boolean value.

v. Assert_when_holds

This method takes a user defined Non – Negated Input Response Oracle as a parameter. The Oracle returns a decision based on the expected behavior of the SUT for valid input. The oracle is defined as a ruby block that consists of the assertion to be tested.

vi. Assert_when_violates

This method receives a user defined Negated Input Response Oracle as a parameter. The Oracle returns a decision based on the expected behavior of the SUT for invalid inputs. The oracle is defined as a ruby block that consists of the assertion to be tested. In addition to the oracle, the method also receives a list of exceptions that may be raised when invalid data is submitted to the SUT.

vii. Run_neg_tests

This method runs negated and non – negated data through both the valid and invalid oracles. The method also reports the results of the tests.

## 2 Exposing the NegTest functionality to MiniTest

In order to expose the NegTest functionality to MiniTest, a new method “given_assumptions” is added to the MiniTest::Assertions module. The “given_assumptions” method takes a block as a parameter. The block consists of a single call of the assert_when_holds and assert_when_violates methods in which the valid and invalid oracles are defined. Below is the general structure of the given_assumptions method.

```
given_assumptions(list_of_assumptions) do
        assert_when_violates do |generated_input|
                valid oracle definition
        end

        assert_when_violates (any_raised_exceptions) do |generated_input|
                invalid oracle definition
        end
end
```

Sequence of steps involved in the execution of a test run.

i. After the user has specified the test using the given_assumptions method as shown above, a NegationController object is created which receives the assumptions and the two oracles.

ii. If a single assumption is provided, the single_assumption method is called. Otherwise the assumption method is called.

iii. Based on the number of assumptions provided, an array of true and false values (truth table ) is generated. For example, a single value would return an array with [True,False]. If two parameters are provided, the array contains all possible combinations for both parameters as shown below:
[ [True,True] , [True,False], [False,True], [False,True] ]

iv. A value is selected from the truth table. For single parameter only one value is selected. For multiple assumptions, a set is selected

v. The is_violated variable which determines if negated(is_violated=True) or non – negated data will be returned is set to a single value from the truth table and then the data generator is queried for a single value based on the assumption.

vi. Negated data is added to a negated data array and non – negated data added to a non negated data array.

vii. The data in both run through both oracles and results reported.

## 3 Adding Custom Generators

To add a custom generator, a class that inherits from the class assumption must be developed. In addition an accessor must be defined to access the class. Both class and accessor must be added to the NegationTest Module. In situations where the existing generators are required, the syntax defined in section 2.4 of Appendix B can be used but only within an accessor. Otherwise the class for the corresponding generator can be used.

To use the custom generator, add the file with the custom generator to the test script using the ruby “require” construct.

# APPENDIX C – Using NegTest

Appendix C gives a description of how the NegTest extension can be used with MiniTest

**USING NEGTEST**

NegTest is an extension of the ruby MiniTest unit testing framework that aids ruby testers to define assumptions about inputs, generate random inputs based on the assumptions and run them against the System Under Test (SUT). This section describes the use of the NegTest extension. A simple example is introduced which is used to describe the extension.

**A simple example**

A simple program input_int_1_10 (input) takes a single string value as input. It must be possible to convert the input into an integer. The program verifies that the value is in the range 1-10. The method returns the value if it is in the range 1-10. Below is the pseudo code for the method:

```
If input is greater than or equal to 1 and less than or equal to 10
            return input
Else
            return "Must be an integer in the range [1, 10]"
```

Following from the above example, a ruby tester might define test cases using MiniTest as shown below:

**Positive tests**

```
 def test_int_1
   assert_equal 1, input_int_1_10(1)
  end

  def test_int_2
   assert_equal 2, input_int_1_10(2)
  end
```

**Negative tests**

```
def test_int_0
   assert_equal "Must be an integer in the range [1, 10]", input_int_1_10(0)
  end

  def test_int_11
   assert_equal "Must be an integer in the range [1, 10]", input_int_1_10(11)
  end

def test_string_11
   assert_equal "Must be an integer in the range [1, 10]", input_int_1_10("11")
  end
```

The tests above are not exhaustive but are an example of the tests expected to be generated by the NegTest extension.

## Negation Test case

Naik and Tripathy [1] describe a test case as a pair of input and expected outcomes from the SUT. A test case more specifically consists of an initialization of test data if required, a call to the system under test and a decision whether the test succeeds or not also known as an oracle. A negation test case consists of the same components which are described below.

## Initialization

This involves the definition of the assumptions about the inputs to be submitted to the SUT. The NegTest extension provides methods for defining the assumptions as ruby Proc objects which are blocks of code bound to a set of local variables and can be accessed in different contexts. The tester defines assumptions that cause the system to exhibit the expected behavior which are valid inputs

The simple example states that the method takes a string value that can be converted to an integer and is in the range 1 <= x <=10. Using the NegationTest extension, this can be defined in the following ways:

i. Using String assumptions

*Proc.new{string(set("1","2","3","4","5","6","7","8","9","10"))}*

This states that the string values 1 to 10 are the only valid values. Negating this assumption results in any string not in the set defined. A string regular expression can also be used to define the assumption.

Because ruby is not typed, it is possible for the assumption to be defined in terms of non string values as shown below:

ii. Using Integers of FixNum

*Proc.new{integer(range(1..10))}*

This states that integer values in the range 1 to 10 are valid values. Negating the assumption above results in any integer not in the range 1-10

iii. Using Floats

*Proc.new{float(range(1..10))}*

This states that float values in the range 1 to 10 are valid values. Negating the assumption above results in any float not in the range 1-10

Any of the assumptions defined above will cause the random generator to generate valid and invalid data. NegTest has methods that return random strings, numbers, Boolean and nil values. In addition, ranges and sets can be defined for the string and number generators. Strings can also be defined using regular expressions.

**Call to the SUT**

The call to the SUT involves stating the SUT and the test input value. For the simple example, the call to the SUT is:

```
input_int_1_10 (generated_input)
```

The generated_input is a random value generated by the random generator based on the assumption defined.

**The Oracle**

NegTest requires the definition of a valid and an invalid oracle. The Non-Negated Input Response Oracle returns a decision based on the expected behavior of the SUT for valid input. For the simple example, the oracle checks that the value returned by the SUT is the same as the integer representation of the input generated by the random generator as shown below:

```
assert_when_holds do|generated_input|
        assert_equal generated_input.to_i, input_int_1_10(generated_input)
end
```

The Negated Input Response Oracle returns a decision based on the expected behavior of the SUT for invalid inputs. For the simple example, the oracle checks for the error message returned when invalid data is provided to the SUT.

```
assert_when_violates do | generated_input |
        assert_equal "Must an integer in the range [1, 10]",
input_int_1_10(generated_input)
end
```

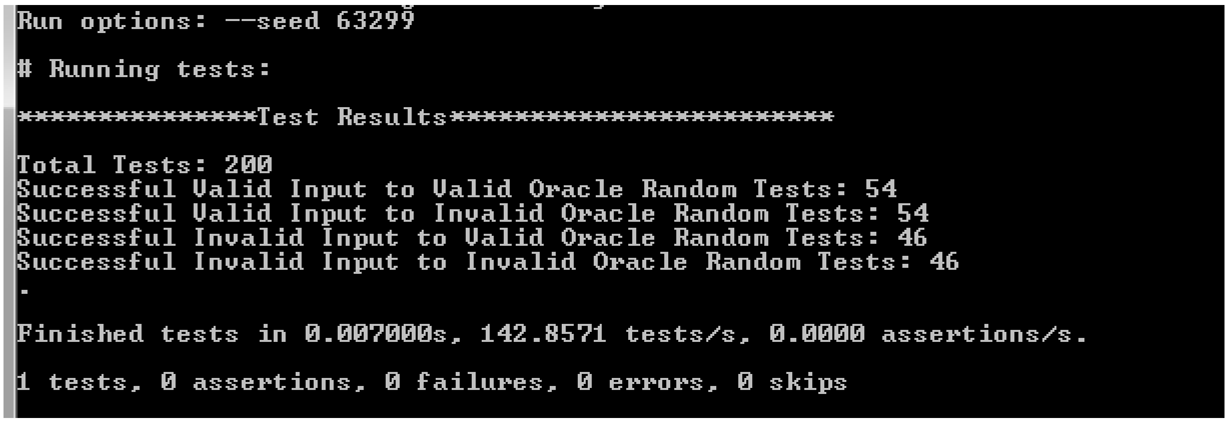
When the tests are run, the oracle carries out four different kinds of tests A, B, C and D that are described in section 7.4.2

The resultant Negation test case is shown below:

```
def test_input_int_1_10
        a=Proc.new{integer(range(1..10))}

        given_assumptions(a) do

                assert_when_holds  do | generated_input |
                assert_equal generated_input.to_i, input_int_1_10(generated_input)
                end

                assert_when_violates do | generated_input |
                assert_equal  "Must  an  integer  in  the  range  [1,  10]",
t.input_int_1_10(generated_input)
                end
        end
end
```

## Results

The data generator creates 100 random inputs that are either valid or invalid based on the assumption provided. Since the inputs have to be tested with the Non-Negated Input Response Oracle and Negated Input Response Oracle, a maximum of 200 tests can be run. The results from the test defined above are shown in the screen shot below

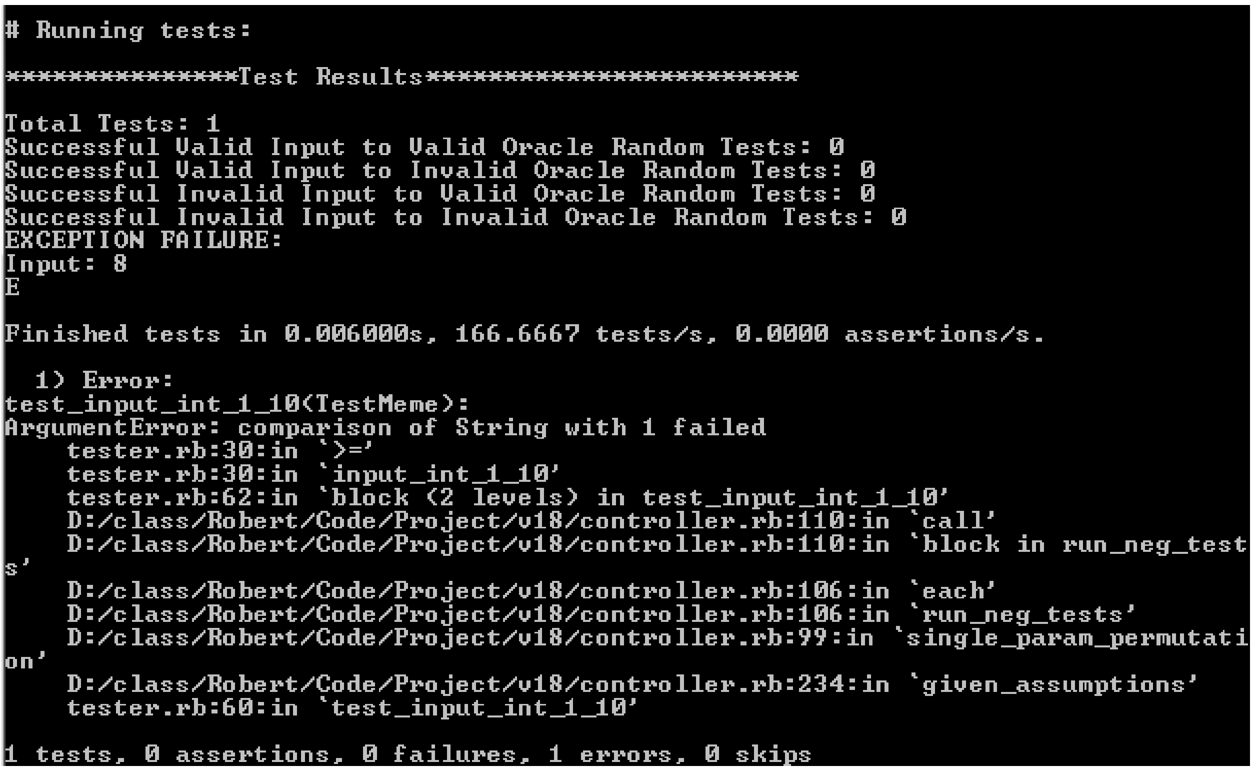


```
Run options: --seed 63299

# Running tests:

***************Test Results*************************

Total Tests: 200
Successful Valid Input to Valid Oracle Random Tests: 54
Successful Valid Input to Invalid Oracle Random Tests: 54
Successful Invalid Input to Valid Oracle Random Tests: 46
Successful Invalid Input to Invalid Oracle Random Tests: 46
.

Finished tests in 0.007000s, 142.8571 tests/s, 0.0000 assertions/s.

1 tests, 0 assertions, 0 failures, 0 errors, 0 skips
```

Screen Shot 1: Results from integer test

The results show 54 tests of valid inputs against each oracle. This also means that the generator created 54 valid inputs. The results also show 46 tests of invalid inputs against each oracle. This also means that the generator created 46 invalid inputs.

The results above which are based on the assumption in section 1.2 (ii) do not show any failure and hence cannot reveal the bug in the program pseudo code. However, using assumption in section 1.2 (i), the bug can identified as shown below:



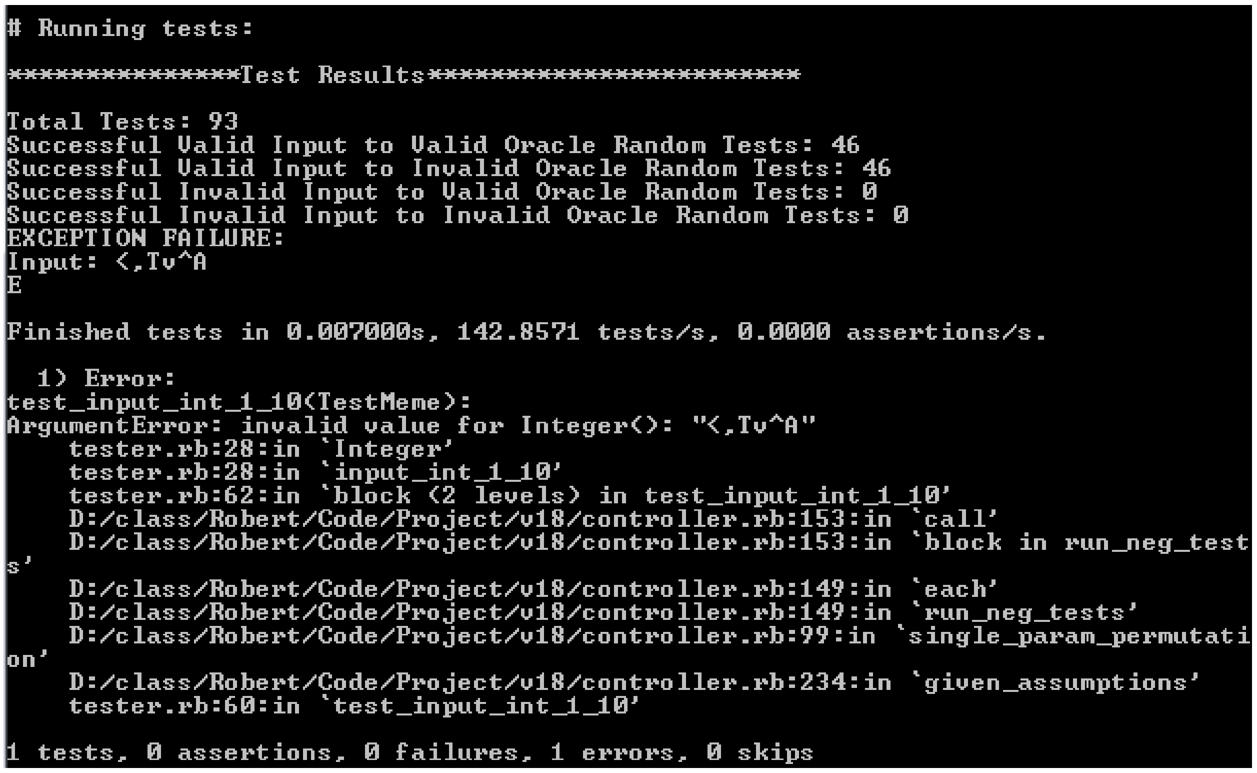
```
# Running tests:

***************Test Results************************

Total Tests: 1
Successful Valid Input to Valid Oracle Random Tests: 0
Successful Valid Input to Invalid Oracle Random Tests: 0
Successful Invalid Input to Valid Oracle Random Tests: 0
Successful Invalid Input to Invalid Oracle Random Tests: 0
EXCEPTION FAILURE:
Input: 8
E

Finished tests in 0.006000s, 166.6667 tests/s, 0.0000 assertions/s.

  1) Error:
test_input_int_1_10(TestMeme):
ArgumentError: comparison of String with 1 failed
    tester.rb:30:in `>='
    tester.rb:30:in `input_int_1_10'
    tester.rb:62:in `block (2 levels) in test_input_int_1_10'
    D:/class/Robert/Code/Project/v18/controller.rb:110:in `call'
    D:/class/Robert/Code/Project/v18/controller.rb:110:in `block in run_neg_test
s'
    D:/class/Robert/Code/Project/v18/controller.rb:106:in `each'
    D:/class/Robert/Code/Project/v18/controller.rb:106:in `run_neg_tests'
    D:/class/Robert/Code/Project/v18/controller.rb:99:in `single_param_permutati
on'
    D:/class/Robert/Code/Project/v18/controller.rb:234:in `given_assumptions'
    tester.rb:60:in `test_input_int_1_10'

1 tests, 0 assertions, 0 failures, 1 errors, 0 skips
```

**Screen Shot 2: Results revealing absence of conversion bug**

The results show that one test was run that resulted in an exception for string input 8. The results also show that an argument error occurred when attempting to compare a string with 1. The results include a trace to the section of the code that caused the error.

From this error we can deduce that the string input is not being converted to an integer and hence the bug. Correcting this bug by introducing the conversion will enable tests equivalent to the total number of valid inputs to be run successfully.

Invalid inputs that cannot be converted to integers will raise exceptions as shown in the screenshot below



```
# Running tests:

***************Test Results************************

Total Tests: 93
Successful Valid Input to Valid Oracle Random Tests: 46
Successful Valid Input to Invalid Oracle Random Tests: 46
Successful Invalid Input to Valid Oracle Random Tests: 0
Successful Invalid Input to Invalid Oracle Random Tests: 0
EXCEPTION FAILURE:
Input: <,Tv^A
E

Finished tests in 0.007000s, 142.8571 tests/s, 0.0000 assertions/s.

  1) Error:
test_input_int_1_10(TestMeme):
ArgumentError: invalid value for Integer(): "<,Tv^A"
    tester.rb:28:in `Integer'
    tester.rb:28:in `input_int_1_10'
    tester.rb:62:in `block (2 levels) in test_input_int_1_10'
    D:/class/Robert/Code/Project/v18/controller.rb:153:in `call'
    D:/class/Robert/Code/Project/v18/controller.rb:153:in `block in run_neg_test
s'
    D:/class/Robert/Code/Project/v18/controller.rb:149:in `each'
    D:/class/Robert/Code/Project/v18/controller.rb:149:in `run_neg_tests'
    D:/class/Robert/Code/Project/v18/controller.rb:99:in `single_param_permutati
on'
    D:/class/Robert/Code/Project/v18/controller.rb:234:in `given_assumptions'
    tester.rb:60:in `test_input_int_1_10'

1 tests, 0 assertions, 0 failures, 1 errors, 0 skips
```

Screen Shot 3: Results revealing unhandled exceptions

To correct the error such as the one above, the tester must modify the SUT to handle exceptions after which all tests will be run successfully.

## Another simple example

This example demonstrates some of the other features of the NegTest Extension. A simple program check_divide_by_zero, takes two integers, a and b, and divides a by b. The specification does not include a check for b not being equal to zero and hence is a source of a failure. Below is the pseudo code for the program that excludes the check of b being equal to zero.

```
If (a and b are integers)
        x= a / b
        return x
else
        return "The input values must be integers"
```

The assumptions for a and b can simply be defined as shown below:

**Pair A**

```
        a=Proc.new{integer()}
        b=Proc.new{integer()}
```

**Pair B**

```
        a=Proc.new{integer()}
        b=Proc.new{integer(set(0,1,2,3))}
```

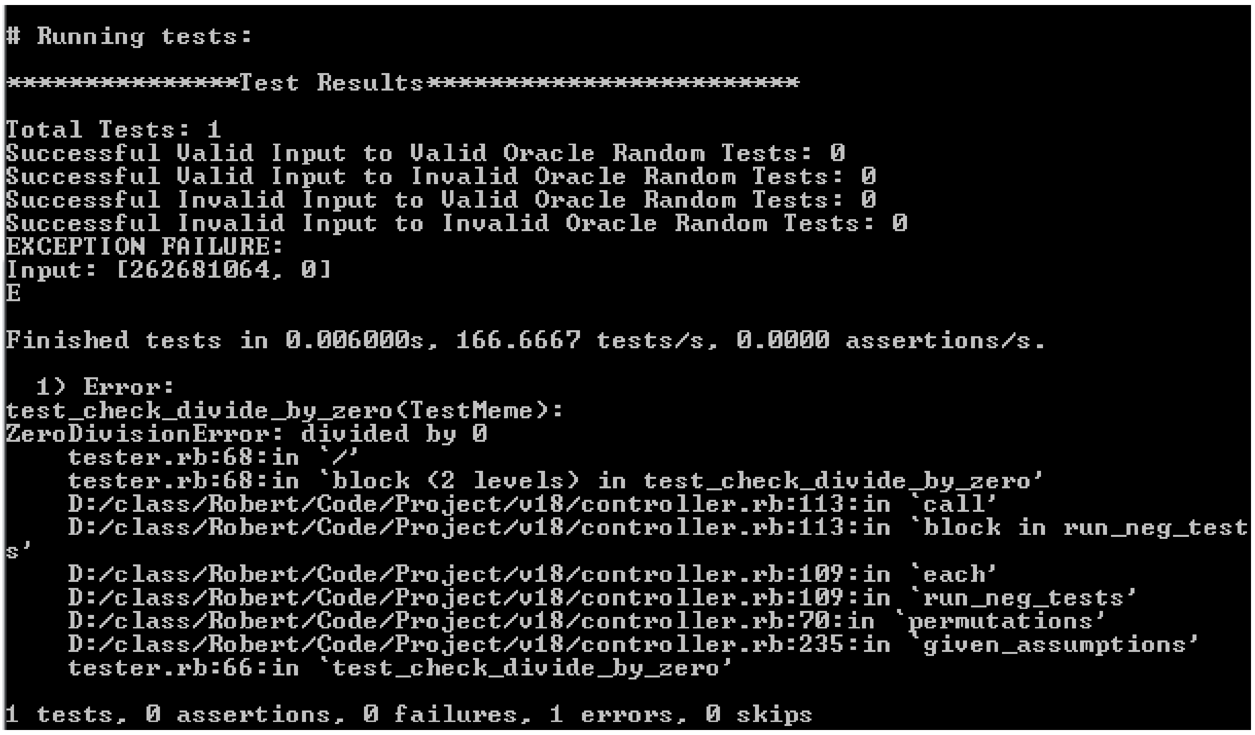
Assumptions in Pair A consider all integers from the smallest integer (-1,073,741,823) to the largest integer (1,073,741,823) to be valid values. The chance of b ever being equal to 0 is one in over 2 million tests cases. So assumption b in Pair A has lower chances of revealing the error.

In assumption b of Pair B, b has a higher chance of revealing the bug and hence a better assumption

The complete specification of the test is shown below:

```
def test_check_divide_by_zero
        a=Proc.new{integer()}
        b=Proc.new{integer(set(0,1,2))}

        given_assumptions(a,b)do
                assert_when_holds do |input|
                        res=0
                        if(input[1] != 0)
                                res=input[0] / input[1]
                        end
                        assert_equal res , check_divide_by_zero(input[0],input[1])
                end

                assert_when_violates do |input|
                  assert_equal "The input values must be integers"
,check_divide_by_zero(input[0],input[1])
                end
        end
end
```

Below is a screen shot of the results from the test



```
# Running tests:

***************Test Results************************

Total Tests: 1
Successful Valid Input to Valid Oracle Random Tests: 0
Successful Valid Input to Invalid Oracle Random Tests: 0
Successful Invalid Input to Valid Oracle Random Tests: 0
Successful Invalid Input to Invalid Oracle Random Tests: 0
EXCEPTION FAILURE:
Input: [262681064, 0]
E

Finished tests in 0.006000s, 166.6667 tests/s, 0.0000 assertions/s.

  1) Error:
test_check_divide_by_zero(TestMeme):
ZeroDivisionError: divided by 0
    tester.rb:68:in `/'
    tester.rb:68:in `block (2 levels) in test_check_divide_by_zero'
    D:/class/Robert/Code/Project/v18/controller.rb:113:in `call'
    D:/class/Robert/Code/Project/v18/controller.rb:113:in `block in run_neg_test
s'
    D:/class/Robert/Code/Project/v18/controller.rb:109:in `each'
    D:/class/Robert/Code/Project/v18/controller.rb:109:in `run_neg_tests'
    D:/class/Robert/Code/Project/v18/controller.rb:70:in `permutations'
    D:/class/Robert/Code/Project/v18/controller.rb:235:in `given_assumptions'
    tester.rb:66:in `test_check_divide_by_zero'

1 tests, 0 assertions, 0 failures, 1 errors, 0 skips
```

**Screen Shot 4: ZeroDivisionError Exception**

The screen shot shows that a single test was run with the value of a=262681064 and b=0 and resulted in a ZeroDivisionError exception. After correcting the error in the specification and the code, the Negated Input Response Oracle needs to be modified to include the new response from the SUT for handling case when b is equal to zero. For example:

```
assert_when_violates do |input|
            assert_includes[ "The input values must be integers",”b cannot be equal to 0”]
                        ,check_divide_by_zero(input[0],input[1])
end
```

In addition the assumption for b needs to be modified to reflect to valid values excluding 0 as shown below:

*b=Proc.new{integer(range([(-1073741823..-1),(1..1073741823)]))}*

b is valid in the range *-1073741823 to -1 and 1 to 1073741823. NegTest allows multiple ranges to be stated for a given input*

At this point, the division by zero error has been resolved and the only errors that can occur are exceptions that occur due types that are not integers. These errors can be resolved by adding the exception handling code for each new exception.

# APPENDIX D – NegTest Evaluation

Appendix D gives a detailed description the evaluation of the NegTest extension with three test candidates.

**Bookland – ISBN.to_isbn_10**

Bookland is a library that provides methods for validating and converting International Standard Book Numbers. The library can be found online at https://github.com/hakanensari/bookland. The Method Under Test (MUT) is the ISBN.to_isbn_10 method whose purpose is to convert ISBN 13 numbers to ISBN 10 numbers.

The method accepts string inputs consisting of 13 numbers. The numbers must start with a 3 digit sequence of 978 or 979. This is followed by 9 numbers. The last number is a checksum which is computed using the formula below:

X13=(10 – (x1+3x2+x3+3x4+…+x11+3x12)mod 10)mod 10

If  X13 == 10 then X13= 0

Examples of valid ISBN 13 numbers include:

```
9780802409430
9781891595240
```

The method first ensures that a valid ISBN 13 number is provided returning a *Bookland::InvalidISBN* exception if an invalid number is provided. The method then attempts to convert the ISBN 13 number into an ISBN 10 number. The method returns the ISBN 10 number.

To test this method using NegTest extension, a custom generator was defined to be able to return the valid ISBN 13 numbers and negated data of any other type and format that does not match the ISBN 13. The method accessor is called isbn_number and is used to state the assumption as shown below

Below is the test that is run in minitest. The test oracles below include a method ISBN10.valid? which was verified to return true when a valid ISBN 10 number is supplied.

```
def test_isbn_to_isbn10
        a=Proc.new{isbn_number()}
        given_assumptions(a)do
                assert_when_holds do |input|
                assert_equal
true,(Bookland::ISBN10.valid?(Bookland::ISBN.to_isbn_10 input))
                end
                assert_when_violates(Bookland::InvalidISBN) do |input|
                assert_equal
false,(Bookland::ISBN10.valid?(Bookland::ISBN.to_isbn_10 input))
                end

        end
end
```

The Non-Negated Input Response Oracle expects a value of true to be returned from the test. For the actual test of the to_isbn_10 method, the value returned from the test is verified using the ISBN10.valid? method which checks if the number returned is a valid ISBN 10 number. The method returns true for a valid ISBN 10 number and false otherwise.

The Negated Input Oracle expects a value of false to be returned from the test. For the actual test of the to_isbn_10 method, the value returned from the test is verified using the ISBN10.valid? method which checks if the number returned is a valid ISBN 10 number. The method returns true for a valid ISBN 10 number and false otherwise.

In addition, the Negated Input Oracle expects an InvalidISBN exception to be thrown for invalid string representations of ISBN 13 numbers.

The ISBN function is then tested with the four test types introduced In section 7.4.2.

Test A passes when a valid ISBN 13 input to the to_isbn_10 method returns a valid ISBN 10 number. This number will return true from the ISBN10.valid? method. The Non- Negated Input Response Oracle returns true. The test fails otherwise and this would mean that the to_isbn_10 method cannot convert a valid ISBN 13 number to a ISBN 10 number.

Test B passes when a valid ISBN 13 input to the to_isbn_10 method returns a valid ISBN 10 number. This number will return true from the ISBN10.valid? method. The Negated Input Response Oracle returns false. The test fails otherwise and this would mean that the to_isbn_10 method cannot convert a valid ISBN 13 number to a ISBN 10 number.

Test C passes when an invalid input to the to_isbn_10 method raises an InvalidISBN exception. The Non- Negated Input Response Oracle returns false. The test fails otherwise and this would mean that either the to_isbn_10 method can convert an invalid input in to an ISBN 10 number or some other value is returned.

Test D passes when an invalid input to the to_isbn_10 method raises an InvalidISBN exception. The Negated Input Response Oracle returns true. The test fails otherwise and this would mean that either the to_isbn_10 method can convert an invalid input in to an ISBN 10 number or some other value is returned.

The table below shows the results of 10 test runs on the ISBN.to_isbn_10? method. The table shows the time for each test, the error identified , the input or type that caused the error, the test type where the failure occurred, and the number of successful tests for each test type A, B, C and D which are described in section 6.4

| Test # | Time (seconds) | Error | Input / DataType | Failed Test Type | A | B | C | D | Tests Run |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.020001 | NoMethodError Exception | boolean | C | 48 | 48 | 0 | 0 | 97 |
| 2 | 0.021001 | NoMethodError Exception | boolean | C | 53 | 53 | 0 | 0 | 107 |
| 3 | 0.020001 | NoMethodError Exception | boolean | C | 52 | 52 | 0 | 0 | 105 |
| 4 | 0.017001 | NoMethodError Exception | Fixnum | C | 36 | 36 | 0 | 0 | 73 |
| 5 | 0.019002 | NoMethodError Exception | boolean | C | 48 | 48 | 0 | 0 | 97 |
| 6 | 0.019001 | NoMethodError Exception | Float | C | 42 | 42 | 0 | 0 | 85 |
| 7 | 0.021001 | NoMethodError Exception | Fixnum | C | 58 | 58 | 0 | 0 | 117 |
| 8 | 0.020001 | NoMethodError Exception | Fixnum | C | 50 | 50 | 0 | 0 | 101 |
| 9 | 0.018001 | NoMethodError Exception | nil | C | 49 | 49 | 0 | 0 | 99 |
| 10 | 0.021001 | NoMethodError Exception | boolean | C | 53 | 53 | 0 | 0 | 107 |

The results show that 100% of the tests of type A pass the test returning a valid ISBN 10 number. 100% of the tests of type B pass the tests returning a valid ISBN 10 number.

However, 100% of the tests of type C fail. The results show that non string types cause a NoMethodError exception. Close analysis of the exception details shows that the application fails when trying check that the invalid value is a valid ISBN 13 number. The method attempts to call a "match" method which is not defined for none string types. This is an indication that no input validation control structures were put in place to ensure that only string types can be used as inputs. In addition it indicates that no exception handling controls were put in place to handle the NoMethodError exception.

The results do not show any exception being caused by a string value. In an attempt to discover if other string representations of numbers can cause the application to fail, the assumption specification was updated to include the NoMethodException. By doing this, tests of type C and D can be run for all invalid string representations of ISBN 13 numbers.

After running the tests again, 100 % of the tests of type C passed. The test showed that the invalid input caused the SUT to correctly raise an InvalidISBN error.

100% of the tests of type D pass. This means that the invalid string inputs correctly raise the InvalidISBN error.

**Chronic – Chronic.parse**

Chronic is a date and time parser. The library can be found online using the url https://github.com/mojombo/chronic. The method under test, parse, accepts various forms of inputs representing dates and times which can be viewed using the link provided above.

To test the method, a single string representation 'tomorrow' was used. The keyword 'tomorrow' will cause the library to return the next day details while using the current date as a start date. The method returns an empty string or a nil value when it fails parse the input value to generate the correct date.

For this test, no custom generator is required. Instead, the string generator is used to always return the keyword 'tomorrow' for no-negated inputs and any other data for negated inputs.

Below is the test defined in MiniTest using NegTest

```
def test_parse_tomorrow

        a=Proc.new{string(set('tomorrow'))}

        given_assumptions(a)do

                assert_when_holds do |input|
                        assert_equal
Date.today+1,(((Chronic.parse(input)).to_s == "") ? "" :
                        Date.parse((Chronic.parse(input)).to_s))
                        end

                assert_when_violates do |input|
                                assert_includes
["",nil],(Chronic.parse(input))
                        end
                end

end
```

The Non- Negated Input Response Oracle expects a value equivalent to tomorrows date to be returned from the test. For the actual test of the Chronic.parse method, the value returned from the test is converted to a Date object and its string representation returned. The oracle returns true if tomorrows date is returned.

The Negated Input Response Oracle expects an empty string or a nil value to be returned from the test of the Chronic.parse. The oracle returns true when a nil or empty string are returned.

The Chronic.parse method is tested with the generated data and the four tests A,B,C and D introduced in section 7.4.2 are run.

Test A passes when the equivalent of tomorrows date is returned for valid input to the Chronic.parse method. The Non- Negated Input Response Oracle returns true. The test fails otherwise and this would mean that the Chronic.parse method cannot return tomorrows date when an input with keyword ‘tomorrow’ is provided.

Test B passes when the valid input ‘tomorrow’ is provided to Chronic.parse method returns tomorrows date. The Negated Input Response Oracle returns false as there is no match to nil and an empty string. The test fails otherwise and this would mean that the Chronic.parse method cannot return tomorrows date when an input with keyword ‘tomorrow’ is provided.

Test C passes when an invalid input to the Chronic.parse method returns an empty string or nil value. The Non- Negated Input Response Oracle returns false. The test fails otherwise and this would mean that either the Chronic.parse method can return tomorrows date or some other value when supplied with invalid input.

Test D passes when an invalid input to the Chronic.parse method returns an empty string or nil value. The Negated Input Response Oracle returns true. The test fails otherwise and this would mean that either the Chronic.parse method can return tomorrows date or some other value when supplied with invalid input.

The table below shows the results of 10 test runs on the Chronic.parse method. The table shows the time for each test, the error identified , the input or type that caused the error, the test type where the failure occurred, and the number of successful tests for each test type A, B, C and D which are described in section 6.4

| Test # | Time | Errors | Input | Failed Test Type | A | B | C | D | Tests Run |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.417024 | | | | 46 | 46 | 54 | 54 | 200 |
| 2 | 0.429024 | | | | 54 | 54 | 46 | 46 | 200 |
| 3 | 0.428025 | | | | 54 | 54 | 46 | 46 | 200 |
| 4 | 0.399022 | | | | 40 | 40 | 60 | 60 | 200 |
| 5 | 0.418024 | | | | 49 | 49 | 51 | 51 | 200 |
| 6 | 0.424025 | | | | 52 | 52 | 48 | 48 | 200 |
| 7 | 0.414023 | Returned the current date | iX,,6a | D | 47 | 47 | 50 | 49 | 194 |
| 8 | 0.414024 | | | | 49 | 49 | 51 | 51 | 200 |
| 9 | 0.273016 | Returned the current date | 4,u444 | D | 37 | 37 | 22 | 21 | 118 |
| 10 | 0.416024 | | | | 45 | 45 | 55 | 55 | 200 |

The results show that 100% of the tests of type A, B and C pass. However tests of type D reveal errors in the MUT. The string values “iX,,6a” and “4,u444” return the current date. The invalid oracle expected the method to return a nil or empty string.

The error detected is of significance because it means all the keywords are subject to this failure or bug that can go unnoticed. The error was not detected in Test C because of the definition of the oracle that was expecting tomorrows date.

**Versionomy – Versionomy.create**

Versionomy is a version number library that creates, manipulates, parses and compares version numbers of various forms. It can be found online at https://github.com/dazuma/versionomy

The method under test is Versionomy.create which returns an object of type versionomy. The method accepts a hash table of symbols or an ordered array of values. The initial assumptions about the inputs are:

The version number created consists of 7 parts that accept the following types of data.

:major, :minor, :tiny, :tiny2, patch_number, patch_number_minor – integers > 0, nil and false which return 1. Any other values will return 0

:releasetype - integers > :alpha, :beta, :final, nil

The specification create method was not clear and hence complete assumptions could not be specified.

To test the create function, the simplest form of a version number consisting of the major, minor, tiny, tiny2 and release type sections is tested. Below is the tests run in MiniTest using the NegTest extension.

```
def test_create_without_patchlevel
		major=Proc.new{integer(range(1..INTEGERMAX))}
		minor=Proc.new{integer(range(1..INTEGERMAX))}
		tiny=Proc.new{integer(range(1..INTEGERMAX))}
		tiny2=Proc.new{integer(range(1..INTEGERMAX))}
		release=Proc.new{string(set(:beta,:alpha,:final,nil))}

		given_assumptions(major,minor,tiny,tiny2,release)do
			assert_when_holds do |input|
				assert_equal true,(Versionomy.create(:major => input[0],
:minor => input[1], :tiny => input[2],:tiny2=>input[3],:release_type=>input[4]).major) > 0
				end

				assert_when_violates(Versionomy::Errors::IllegalValueError)
do |input|

				assert_includes [0],(Versionomy.create(:major => input[0],
:minor => input[1], :tiny => input[2],:tiny2=>input[3],:release_type=>input[4]).major)

				end
			end

		end
```

The Non- Negated Input Response Oracle expects true to be returned from the test. For the actual test of the Versionomy.create method, an object is created and the major component of the object is checked to ascertain that it is an integer that is greater than zero. The oracle returns true if the major portion is indeed an integer greater than zero.

The Negated Input Response Oracle expects 0 to be returned from the test of the Versionomy.create method. The oracle returns true when a 0 is returned.

The Versionomy.create function is then run against the four tests A,B,C and D introduced in section 7.4.2

Test A passes when major component returned for valid input to the Versionomy method is an integer greater than zero. The Non- Negated Input Response Oracle returns true. The test fails otherwise and this would mean that the Chronic.parse method cannot return tomorrows date when an input with keyword ‘tomorrow’ is provided.

Test B passes when the valid input ‘tomorrow’ is provided to Chronic.parse method returns tomorrows date. The Negated Input Response Oracle returns false as there is no match to nil and an empty string. The test fails otherwise and this would mean that the Chronic.parse method cannot return tomorrows date when an input with keyword ‘tomorrow’ is provided.

Test C passes when an invalid input to the Chronic.parse method returns an empty string or nil value. The Non- Negated Input Response Oracle returns false. The test fails otherwise and this would mean that either the Chronic.parse method can return tomorrows date or some other value when supplied with invalid input.

Test D passes when an invalid input to the Chronic.parse method returns an empty string or nil value. The Negated Input Response Oracle returns true. The test fails otherwise and this would mean that either the Chronic.parse method can return tomorrows date or some other value when supplied with invalid input.

The table below shows the results of 10 test runs on the Versionomy.create method. The table shows the time for each test, the error identified , the input type, the test type where the failure occurred, and the number of successful tests for each test type A, B, C and D.

| Test # | Time | Error | Input | Failed Test Type | A | B | C | D | Tests Run |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.159009 | Negative Number | | C | 3 | 3 | 0 | 0 | 7 |
| 2 | 0.158009 | Floating point numbers | | C | 4 | 4 | 0 | 0 | 9 |
| 3 | 0.155009 | string | qwefsdt(^g | C | 3 | 3 | 0 | 0 | 7 |
| 4 | 0.156009 | float | | C | 1 | 1 | 0 | 0 | 3 |
| 5 | 0.155009 | Negative number | | C | 3 | 3 | 0 | 0 | 7 |
| 6 | 0.157009 | Floating | | C | 2 | 2 | 0 | 0 | 5 |
| 7 | 0.154009 | Floating | | C | 3 | 3 | 0 | 0 | 7 |
| 8 | 0.156009 | string | 01nhO! | C | 4 | 4 | 0 | 0 | 9 |
| 9 | 0.157009 | String | tybungd | C | 3 | 3 | 0 | 0 | 7 |
| 10 | 0.173010 | Negative number | | C | 6 | 6 | 0 | 0 | 13 |

The results show that inputs involving negative numbers, floating point numbers and random strings were returning valid version numbers. From this we can conclude that either the initial assumptions about the inputs are wrong or the library has bugs that need to be resolved.