DESIGN AND EVALUATION OF MUTATION OPERATORS FOR THE ASMETAL LANGUAGE

BY

Osama J. AlKrarha

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

Software Engineering

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Osama Jamil AlKrarha** under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SOFTWARE ENGINEERING.**

____

Dr. Adel Ahmad
Department Chairman

____

Dr. Salam A. Zummo
Dean of Graduate Studies

1/6/14
____

Date

____

Dr. Jameleddine Hassine
(Advisor)

____

Dr. Moataz Ahmed
(Member)

____

Dr. Sami Zhioua
(Member)

To My Parents

ACKNOWLEDGMENTS

All praise and thank to almighty Allah (GOD) for giving me the strength, resolve, health, patience, and knowledge to complete this work.

I acknowledge, with deep gratitude and appreciation, the inspiration, encouragement, valuable time and guidance given to me by DR. Jameleddine Hussine, who served as my major advisor and mentor. Thereafter, I am deeply indebted and grateful to Dr. Moataz Ahmad, and Dr. Sami Zhioua, my committee member, for their extensive guidance, continuous support, and personal involvement in all phases of this research. I am also grateful to my director, Dr. Sadiq Sait their constructive guidance, valuable advice and cooperation.

I also would like to express my deepest gratitude to my mother, father, brothers, sisters, and friends, for their emotional and moral support throughout my academic career and also for their love, patience, encouragement and prayers.

Finally, I would like to thank all the colleagues at the CCITR/Research Institute at King Fahd University for their and support during the period this research took place is greatly acknowledged.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABS	:	Absolute Value Operator
AOR	:	Arithmetic Operator Replacement Operator
ARO	:	Add Rule Operator
ASM	:	Abstract State Machine
ASM SL	:	Abstract State Machine Standard Language
AsmetaL	:	Abstract State Machine Meta Model Language
AsmL	:	Abstract State Machine Language
CDoR	:	Choose DoRule Replacement Operator
CDR	:	Choose Domain Replacement Operator
CIR	:	Choose IfNoneRule Replacement Operator
CLI	:	Command Line Interface
CRE	:	Choose Rule Exchange Operator
CRRO	:	Case Rule Replacement Operator
CTM	:	Constant Term Modification Operator
CTR	:	Constant Term Replacement Operator
CTRO	:	Case Term Replacement Operator

DIR	:	Default Initialization Replacement Operator
DSC	:	Delete Switch Case Operator
EBNF	:	Extended Backus–Naur Form
EDR	:	Extend Domain Replacement Operator
EIR	:	Extend ID Replacement Operator
ENF	:	Expression Negation Fault Operator
ERR	:	Else Rule Replacement Operator
ERRO	:	Extend Rule Replacement Operator
ETR	:	Else Term Replacement Operator
FCRP	:	Forall Choose Rules Permutation Operator
FDoR	:	Forall DoRule Replacement Operator
FQTDR	:	Finite Quantification Term Domain Replacement Operator
FQTP	:	Finite Quantification Terms Permutation Operator
FSM	:	Finite State Machine
FTP	:	Function Type Permutation Operator
GUI	:	Graphical User Interface
ICR	:	Invariant Condition Replacement Operator

IDD	:	Invariant Declaration Deletion Operator
IDE	:	Integrated Development Environment
IIP	:	Initialization ID Permutation Operator
ISD	:	Initialization Statement Deletion Operator
LNF	:	Literal Negation Fault
LOR	:	Logical Operator Replacement Operator
LRR	:	Let Rule Replacement Operator
LRVA	:	Let Rule Variable Assignment Operator
LRVR	:	Let Rule Variable Replacement Operator
LTS	:	Label Transition System
MCDC	:	Multiple Condition Coverage
MRR	:	Main Rule Replacement Operator
MS	:	Mutation Score
RGCR	:	Rule Guard Condition Replacement Operator
ROR	:	Relational Operator Replacement Operator
RRO	:	Replace Rule Operator
RTS	:	Rule to Skip Rule Operator

S2PB	:	Sequential to Parallel Block Operator
SBSDL	:	Sequential Block Statement Deletion Operator
SCP	:	Switch Case Permutation Operator
SSM	:	Sequence Rule Order Permutation Operator
SSSC	:	Stuck Switch to Specific Case Operator
STF	:	Stuck at True False Operator
TGCR	:	Term Guard Condition Replacement Operator
TRR	:	Then Rule Replacement Operator
TTR	:	Then Term Replacement Operator
UOI	:	Unary Operator Insertion Operator

ABSTRACT

Full Name : Osama Jamil AlKrarha

Thesis Title : Design and Evaluation of Mutation Operators for AsmetaL Language

Major Field : Master of Science in Software Engineering

Date of Degree : May, 2014

Abstract State Machines (ASMs) have been introduced by Gurevich in 1984. Abstract State Machines aim to bridge the gap between informal and formal descriptions by transforming informal specifications to clear and concise specifications. ASM Models are simple, concise, and executable. In addition, they support various levels of abstraction, and provide a well-defined refinement models. ASMs support concurrent and non-deterministic specifications. Several ASM-based languages were proposed to develop and validate Abstract State Machines specifications. Asmeta is an interoperable and integrated framework that provides a standardized infrastructure that serves different specific domain tools and languages. Mutation testing is fault-based testing technique aims to assess the adequacy of test suites by introducing errors into program code to reveal the seeded errors. This thesis proposes a mutation based approach to test ASM specifications. A set of mutation operators were designed for AsmetaL language. The proposed AsmetaL-based operators are analyzed and evaluated empirically using several case studies. Furthermore, the proposed set of operators have been implemented in MuAsmetaL, an AsmetaL mutation testing tool, allowing for validation and execution of mutants, as well as the generation of related statistics. As an application of the proposed approach, test suites generated using ATGT, an AsmetaL compatible testing tool implementing various coverage criteria, were assessed. Mutation testing is known for its high computation cost.

In this thesis, both selective and random mutation were applied to AsmetaL mutants resulting in substantial gains in terms of effectiveness and cost savings.

ملخص الرسالة

الاسم الكامل: أسامة جميل القرارعة

عنوان الرسالة: تصميم وتقييم مشغلات الطفرة للغة AsmetaL

التخصص: درجة الماجستير في هندسة البرمجيات

تاريخ الدرجة العلمية: مايو، 2014

استحدثت آلات الحالة المجردة (ASM) بواسطة جورفيتش في عام 1984. وتهدف آلات الحالة المجردة لسد الفجوة بين المواصفات غير الرسمية والرسمية من خلال تحويل المواصفات غير الرسمية لمواصفات رسمية واضحة وموجزة. وتعتبر نماذج ASM بسيطة وموجزة، وقابلة للتنفيذ. بالإضافة إلى أنها تدعم مستويات مختلفة من التجريد، وتوفر نماذج صقل واضحة المعالم. وتدعم ASMs كل من المواصفات المتزامنة وغير القطعية. وقد تم اقتراح عدة لغات على أساس ASM للتطوير والتحقق من صحة مواصفات آلات الحالة المجردة. Asmeta هي عبارة عن إطار للتشغيل المتبادل و المتكامل والتي توفر بنية تحتية موحدة تخدم مختلف لغات وأدوات مجال معين. ويعد اختبار الطفرة تقنية تهدف لتقييم مدى ملاءمة مجموعات الاختبار من خلال تعمد إدخال أخطاء في التعليمات البرمجية للبرنامج وذلك من أجل تقييم مدى قدرة مجموعة الاختبار الكشف عن الأخطاء التى تم إدخالها آنفا. وتقترح هذه الرسالة نهج اختبار الطفرة يستند على تقنية المواصفات ASM. وفي هذه الرسالة، تم تصمم مجموعة من مشغلات الطفرة للغة AsmetaL. وتم تحليل وتقييم هذه المشغلات تجريبيا باستخدام عدة دراسات حالة. وعلاوة على ذلك، فإن مجموعة المشغلات المقترحة تم تنفيذها بواسطة MuAsmetaL، والتي تعتبر أداة لإجراء اختبار الطفرة للغة AsmetaL، مما يسمح للتحقق من صحة وتنفيذ الطفرات، فضلا عن توليد الإحصاءات ذات الصلة. وكتطبيق للنهج المقترح، تم تولد مجموعات اختبار باستخدام أداة ATGT المتوافقة مع لغة AsmetaL بناء على معايير التغطية المختلفة، وجرى تقييمها. ومن المعروف عن اختبار الطفرة أنه ذا تكلفة حسابية عالية. وفي هذه الرسالة، تم تطبيق كل من الطفرة الانتقائية والعشوائية للغة AsmetaL مما أدى لنتائج ايجابية من حيث الفعالية وخفض التكلفة الحسابية.

CHAPTER 1

INTRODUCTION

The demand for high quality software has increased in various fields and disciplines. Therefore, it led to an increased focus on the effectiveness of the processes used in the software industry. Software testing is considered one of the most critical processes that lead to software projects success or failure, therefore, software engineers and researchers in this area aim to put more emphasis on the effectiveness of software testing. Software testing spans the entire software life cycle from requirements stage to the maintenance stage. The magnitude of faults can be reduced if they were detected at the early stages.

1.1 Motivation

The typical way to validate unstructured software specifications is through inspection [1], which is usually carried out manually and takes considerable time and effort. In contrast, the usage of formal specifications reduces such an effort and time, while allowing for automated validation. Abstract State Machines (ASMs) [2] is a formal paradigm that has proved its merit in many fields such as software requirements engineering, network protocols engineering, and system engineering. Handling software requirements using Abstract State Machine overcomes the natural language with the following advantages: Simplicity, precise semantics, various levels of abstractions, and executability. In addition, it provides a well-defined validation and verification model. Moreover, ASM Models can be used to generate portions of the implementation.

Mutation testing technique is a fault-based technique that has been successfully used to test various programming and specification languages. This thesis introduces a new ASM-based mutation testing approach to assess the adequacy of ASM test suites.

1.2 Problem Statement

The goal of this research is to develop a mutation testing approach for AsmetaL, an ASM-based language. The proposed approach would allow both practitioners and researchers to assess and improve the adequacy of AsmetaL test suites. The main goal is decomposed into the following sub-goals:

- *Sub-Goal 1:* Definition of a set of mutation operators for AsmetaL as a concrete incarnation of ASM mutation operators.
- *Sub-Goal 2:* Investigation of the applicability of the proposed mutation operators to various case studies.
- *Sub-Goal 3:* Assessment of the effectiveness of the designed operators.
- *Sub-Goal 4:* Investigation the applicability of cost reduction techniques such as selective and random mutation in the context of the AsmetaL language.
- *Sub-Goal5:* Develop an AsmetaL mutation testing tool that allows for validation and execution of mutants and the generation of mutation related statistics.

1.3 Research Hypothesis

The research hypotheses can be formulated as follows:

Research Hypothesis 1:

Our first research hypothesis is denoted as follows:

> "Mutation testing can be applied to the Abstract State Machines (ASM) formalism. This can be achieved through the design and the application of ASM-based mutation operators."

Research Hypothesis 2:

Our second research hypothesis is denoted as follows:

> "ASM-based mutation testing is an effective approach to assess the adequacy of ASM-based test suites."

Research Hypothesis 3:

Our Third research hypothesis is denoted as follows:

> "Mutation-based testing cost reduction techniques, such as selective and random mutation can be applied in the context of Abstract State Machines specifications."

1.4 Thesis Approach

Mutation testing has been successfully applied to many programming and specification languages. In this thesis, we investigate the application of the mutation testing approach to the ASM-based specification language AsmetaL.

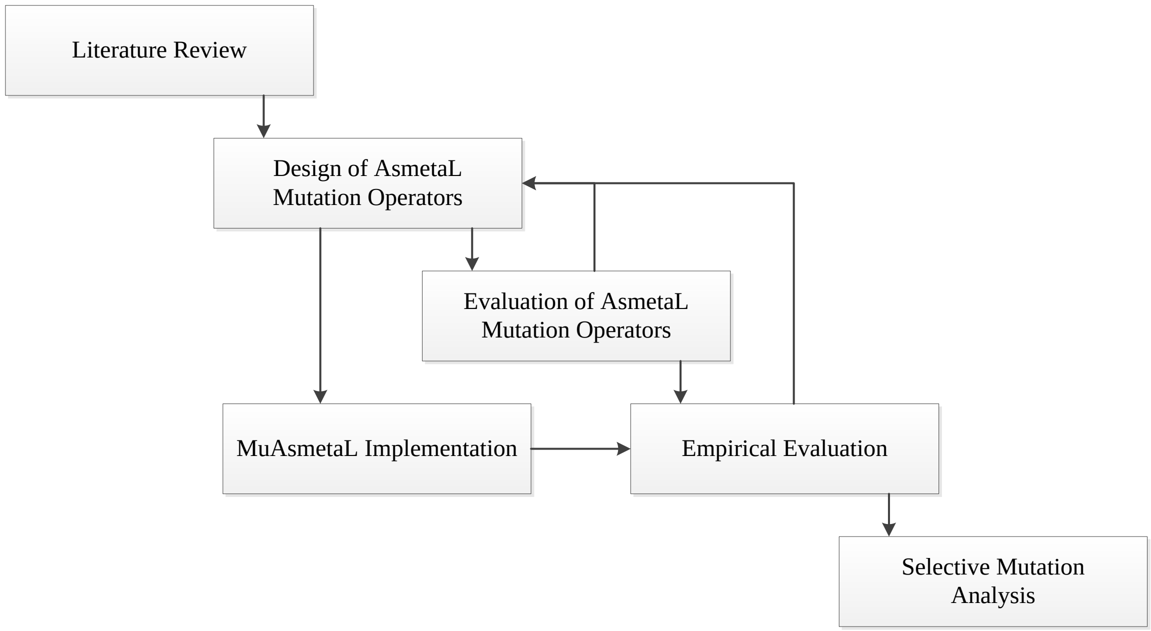


Figure 1: Thesis tasks workflow

As shown in Figure 1, this thesis includes, the design and evaluation of mutation operators for AsmetaL, the implementation of mutation operator for AsmetaL. In addition, these operators will be evaluated empirically using several case studies. Finally, cost reduction techniques such as selective-mutation and random mutation are investigated in the context of the AsmetaL language.

1.5 Thesis Contributions

This thesis offers four main contributions

1.5.1 Contribution 1: Design and Evaluation of Mutation Operators for the AsmetaL Language

We have proposed a set of 49 operators for the AsmetaL language. The resulting operators are categorized into 5 categories targeting different types of AsmetaL faults. Each mutation operator is described using a concrete example and analyzed with respect to the produced mutants (*e.g.,, valid/invalid, equivalent/non-equivalent, etc*.). Furthermore, a mathematical characterization of the upper bound of the number of generated mutants is provided for each operator. Chapter 4 presents and discusses the set of proposed AsmetaL-based mutation operators.

1.5.2 Contribution 2: Empirical Evaluation of the Proposed Approach

Our proposed mutation-based approach is evaluated empirically using a set of 7 case studies of different sizes. We have shown that mutation testing can be applied effectively to ASM-based specifications. Furthermore, as an application of the proposed approach and since the only tool, spotted in the literature, that supports the generation of test cases for AsmetaL language is ATGT, we have focused on the evaluation of the test suites produced using the ATGT coverage criteria. We have shown that some ATGT coverage criteria are more adequate than others are. Chapter 6 presents and discusses our empirical experiments.

1.5.3 Contribution 3: Development of MuAsmetaL

We have developed a prototype tool (*called MuAsmetaL*) to perform AsmetaL-based mutation testing. The tool presents many features that can be summarized as follows:

- Generating mutants based on the proposed operators.
- Validating the correctness of all the generated mutants using AsmetaLc.
- Validating syntactic equivalency of generated mutants against the original specification.
- Running test cases against the original specification.
- Running test cases against mutants.
- Calculating mutation score per operator and for all mutants.

Chapter 5 presents our MuAsmetaL tool.

1.5.4 Contribution 4: Investigation of Cost Reduction Techniques in the ASM Context

Mutation testing is known to have a high computation cost due to the large number of generated mutants. Many techniques have been proposed to reduce the cost of the application of mutation testing. In this thesis, we have applied random mutation and selective mutation to AsmetaL specifications. As discussed in Chapter 7 , we were able to achieve satisfactory results with respect to the resulting mutation score and the cost savings.

1.6 Issues not Addressed in this Thesis

This thesis will not address the following issues:

- Detection of equivalent mutants: we haven't proposed any technique to preform mutation equivalency analysis.
- Generation of test cases: The proposed approach aims at providing a useful adequacy analysis technique to assess test suite for AsmetaL language. However, it does not provide a mechanism to generate test cases.
- Higher order mutation testing: Only single order mutation testing will be addressed in our approach.
- Applying mutation testing to non-deterministic specifications is out of the scope of this thesis.

1.7 Thesis Outline

The remaining parts of the thesis are divided into eight chapters:

Chapter 2: provides the general background information that sets the stage for our proposed approach. It consists of two parts. The first part introduces the background information about the basic concepts, notations, and technologies about Abstract State Machines (ASM) paradigm. The second part presents the basic definitions of mutation-based testing methodology.

Chapter 3: provides an overview of the state of art for testing Abstract State Machines. In addition, it includes a brief overview of formal specification (*e.g., FSM, State chart, etc.*) mutation testing approaches and techniques.

Chapter 4: provides an in-depth look at our proposed approach including methodology, mutation testing operators, empirical evaluation, developed tool, and selective mutation criteria.

Chapter 5: presents an overview of the MuAsmetaL (*a tool for mutating AsmetaL syntax, developed as a proof of concept*) including tool requirements, architecture, screenshots, and tool limitations.

Chapter 6: provides an empirical evaluation of our proposed approach aiming to assess the effectiveness of the proposed AsmetaL mutation operators. Several case studies adopted from the literature were used in the experiment.

Chapter 7: applies random mutation and selective mutation to AsmetaL specifications.

Chapter 8: recalls the contributions of the thesis. This chapter concludes with some directions for future research.

CHAPTER 2

Basic Definitions and Notations

We have to set the stage for our proposed approach by providing a general background information. This chapter consists of two parts. First, an introduction to Abstract State Machines paradigm including the basic concepts, notations, and technologies. Second, an introduction to mutation testing technique including basic definitions and methodology.

2.1 Abstract State Machines

2.1.1 ASM Thesis

The concept of Abstract State Machines (ASM) was originally proposed by Gurevich [3] in his thesis work back in 1984 that aims to allow the transformation of any sequential algorithm into an abstract state machine (*referred to as sequential dynamic structure*) in order to mimic any sequential computational devices. According to an Abstract State Machines historical study by Buorger [4], spanning the period from 1984 to 2001, the stages of the evolution of abstract state machines can be classified into four different stages. (i) The early stages where dynamic structure was proposed by Gurevich to simulate any sequential computational devices. (ii) The second stage is when abstract state machines were adopted in the industry, because it provides structural and analytical ability. (iii) The third stage focused on the ability and efficiency of abstract state machines to build, analyze and verify various types of practical applications with various levels of complexity. (iv) The fourth stage, which is the current stage, where the use of abstract state machines in software development is noticed, especially using ground model and stepwise refinement process which have been used in requirements engineering processes.

2.1.2 ASM in a Nutshell

The main idea behind ASMs is to eliminate any ambiguity by transforming informal specifications to clear and formal specifications using a mathematical representation that enforces tractability, reliability, predictability, and quality. Furthermore, ASMs support formal verification, validation, and analysis techniques. The ASM concept is used to simplify the design of complex systems, such as concurrent and reactive systems. In software engineering process, ASM can be applied during the requirements engineering phase, the design phase, and testing phase. ASM-based specifications can be used to assess the quality of software, provide test oracles [5], and automate the generation test suites. Farahbod and Glasser [6] summarized the characteristics of ASMs as follows: i) Simplicity and conciseness. ii) Precision .iii) Variant level of abstraction. iv) Evolutionary iv) Well defined refinement model vi) Executable. vii) Concurrent and non-deterministic. viii) Well defined verification model. The strengths of ASMs are summarized as follows: i) Provides a dynamic structural notation. ii) Simple. iii) General purpose and problem independent. iv) Flexible level of abstraction. vi) Provides a proof of correctness (through tractability).

A basic ASM rule can be described as follows:

if **guard** then **rule1** else **rule2** end if

Where guard is a Boolean condition. Where the rule is a finite set of update function defined by the transform terms of ASM.

A basic ASM function can be described as follows:

$$f : (t1; t2, \ldots, tn)$$

There are two types of ASM functions:

a. Static functions that are not updated during the run time.
b. Dynamic functions that can be classified into four types: i) Controlled: updated only by rules ii) Monitored: updated by the environment iii) Interaction: updated by the rules and by the environment iv) Derived function that are neither updated by rules nor by the environment.

Transition Rules

ASM provides seven types of rules:

1. Skip Rule: do nothing.
2. Update Rule: while in next state value of *f* is updated to *S*.
3. Block Rule: *R* and *S* are executed in parallel.
4. Conditional Rule:

if *g* **the** *R* **else** *S*

If *g* is true, execute *R*, otherwise excuse *S*.

5. Let Rule:

Let *x* **=** *t* **in** *R*

Assign value of *t* to *x* and execute *R*.

6. Forall Rule:

forall x **with** g **do** R

Execute R in parallel for each x that satisfies the condition g.

7. Call Rule:

$$r(t_1; t_2, \ldots, t_n)$$

Call r with parameters $t_2, \ldots, t_n$.

ASM Types

Sequential ASMs referred to as ASMs that execute sequential time in a step-by-step manner, with non-empty set of sates, non-empty set of initial states and one step transformation function while closed under isomorphism [7]. It is proven that for every sequential algorithm, there exists a behaviorally equivalent sequential ASMs [8]. Parallel ASM is referred to ASMs that execute in sequential global time and have the ability to create new parallel components on-the-fly [9]. For every parallel algorithm, it is proven that must exist a parallel ASM that is behaviorally equivalent. Distributed ASM consists of finitely many single agents sequential ASMs in which it has finitely many predecessors, every agent are linearly ordered, and each finite initial segment corresponds to a state.

2.1.3 ASM Languages

Many languages were developed as incarnation of ASMs concept, in this subsection, we present few of them.

1. AsmL (Abstract State Machine Language) [10]

The AsmL [11] language was developed by Microsoft to provide a tool that supports the basics of ASM, while being integrated with the Microsoft .Net frameworks. That integration is possible because AsmL is designed to comply with meta-modeling.

In addition, AsmL can be considered as an executable model that supports automatic testing and automatic test case generation. AsmL takes advantage of the well-defined and used FSM testing techniques in order to automate the test case generation and evaluation processes as mentioned in section 3.1.5.1. AsmL is equipped with a set predefined of data type beside that it is fully integrated with all elements of the .NET frameworks such as (*e.g.,, interfaces, classes, methods and delegates*). Moreover, both .Net framework languages and AsmL models can call each other natively without any adapter. AsmL supports parallel, sequential, deterministic and nondeterministic ASM specification. ASML has the ability to handle exceptions similarly to other .Net framework languages. Barnett et al. [11] have introduced a model-based testing environment, based on AsmL. This environment takes care of parameter generation, FSM generation, sequences generation, and runtime execution.

2. CoreASM [12]

CoreASM, proposed by Farahbod and Glasser [13], provides all basics of ASM and fulfills all characteristics mentioned in section 2.1.2. The focus of CoreASM is to support extensibility by providing an open source framework offering the basis and foundations for third parity tools (*e.g., model checkers and test generation tools*). It is an extensible language that support the extensibility of both language's syntax and semantics with extensible grammar, extensible engine which provides the ability to extend functionality and control of ASM, extensible simulator that supports multi agents for distributed abstract state machines (*multi ASMs that interact with each other and their environment*) and a library provides additional features. Since it supports extensibility, CoreASM features a micro kernel that support customization based on user needs and domains. However, CoreASM suffers from excessive extensibility, which requires a fast multi grammar parser. In addition, it does not provide predefine modes (*untyped models*).

3. ASM Meta-model [14]

Combining model driven engineering with ASMs concepts, provides another dimension in which it exploits the advantage of meta modeling in term of separation the ASMs specifications from language, tool and environment that have been used to develop it. Moreover, it enforces the ability of model transformation and provides higher interoperability in case of dissimilar languages. According to Gargantini et al. [15] Meta model provides a language independent standardized abstract notation for ASMs with an intuitive graphical representation of ASMs that act as an interchange policy among different ASM tools. In addition, it provides an infrastructure that serves the third party tools and languages based on standard libraries and APIs to support interoperability and integration among tools. One of the main characteristics of meta modeling approach is its readiness for automation.

3.1. Asmeta [16]

Asmeta is an interoperable and integrated framework that provides a standardized infrastructure (*standard libraries, APIs and interchange format*) that serves different specific domain tools and languages [16].

3.2. AsmM [17]

AsmM [17] defines language syntax used to specify ASMs specifications based on Object Management Group OMG framework. AsmM is combined with specific domain description that specifies the creation, access, interchange and manipulation of ASMs.

4. ASM SL and ASM Workbench [18]

ASM workbench [19] is an integrated environment based on ASM specification language that supports five main functionalities: ASM basics functionalities delivered by workbench kernel, type checking provided by model checker component, simulation by simulators, debugging based on debugging GUI and verification provided by model checker. Workbench supports parallel and sequential, in addition to, deterministic and non-deterministic ASM models.

Original ASM specifications do not support neither static nor universal functions; however, ASM Workbench overcomes these issues by deriving these functionalities from ASM specification languages. Moreover, the original specifications of ASM were untyped, while, ASM workbench supports predefined type as mentioned earlier. In addition, ASM workbench is built to be extensible, so that other tools can build on its functions. ASM workbench relies on SMV (*symbolic model verifier*) to preform model checking as shown in Figure 2, where the infinite model of ASM is transformed into finite model based on fitness constraints before being fed into the model checker.

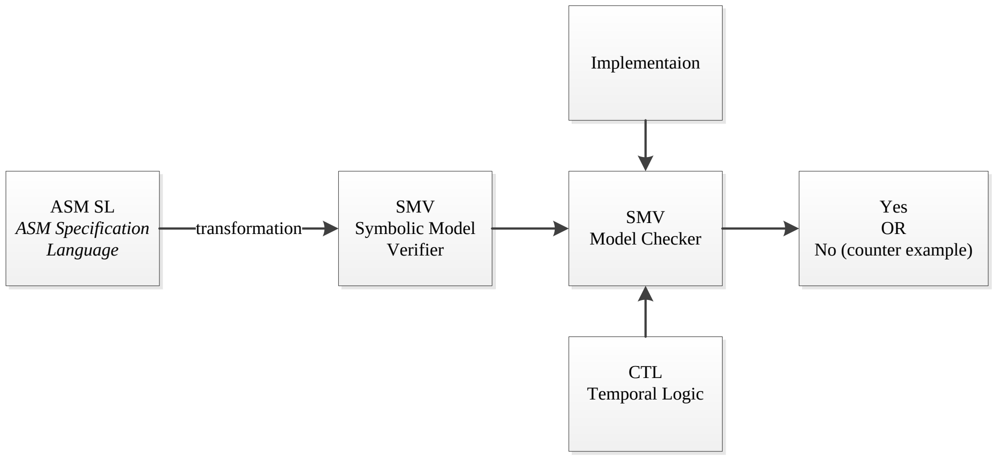


Figure 2: ASM Workbench model verification process

5. Comparison of ASM environments

In order to compare the aforementioned ASMs languages/environments, I have proposed the following attributes:

I. Typed: the availability of predefined data types.

II. Meta-Model: the support of using meta-model.

III. Integration: the ability to integrate with other tools.

IV. Test Generation: the ability to automate test cases generation.

V. Extensible: the ability to extend the environment (*syntax and functionality*).

VI. Infrastructure: offering infrastructure for third party tools.

Table 1 shows the comparison between ASM tools based on the proposed properties.

Table 1: Comparison of ASM Programs

	Typed	MetaModel	Exceptions	Integration	Test Generation	Extensible	Infrastructure
Original ASM	Untyped	No	No	No	No	No	No
SpecExplorer	Typed	Yes	Yes	.Net framework	Yes	No	No
CoreASM	Untyped	No	No	Third party Tools	No	Yes	Yes
Workbench	Typed	No	No	No	No	No	No
Asmeta	Typed	Yes	No	Third party Tools	No	No	Yes

2.1.4 AsmetaL

AsmetaL [16][20], [21][22] consists of four main sections: i) Header section. ii) Body section. iii) Main rule. iv) Initialization section (*optional*). Figure 3 shows the main structure of AsmetaL language and Table 2 provides a simple example of AsmetaL specification. The header section includes three sub sections: i) Import clause is an optional subsection, which identifies any external module that needs to be included, In addition, it allows inclusion of selectable domains, functions, and rules. ii) Export clause is an optional subsection, which identifies all portions of the current module that are permitted to be imported in other modules. iii) Signature is mandatory subsection in which all domains and functions signatures are defined respectively.

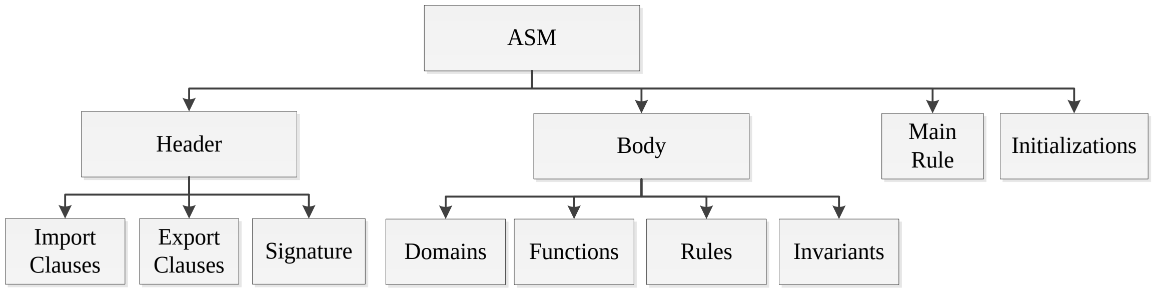


Figure 3: AsmetaL basic structure

Table 2: AsmetaL simple example

Header	`asm example`
	`import ../STDL/StandardLibrary`
	`signature :`
	`monitored value:Integer`
	`out msg:String`

Body	`definitions :`

Main Rule	`main rule r_main =`
	`      if(value>10) then`
	`            msg := "greater than 10"`
	`      else`
	`            msg := "10 or less"`
	`      endif`

Initialization	`default init s0:` `function msg = ""`

Typically as shown in Figure 6, a domain is either a concrete domain, which is a sub domain of other domain, or a type domain. Type domain is either any domain (*the most universal domain, all domains are subset of Any Domain, denoted by* ***any***), structured domain (*product domain, sequence domain, powerset domain, bag domain, and map domain*), enumerator domain, abstract domain, or basic domain as shown in Table 3, Table 4, Table 5, and Table 6.

Table 3: concrete domain signature

(dynamic)? domain ID_DOMAIN subsetof ID_DOMAIN

Table 4: enumerator domain signature

enum domain ID_DOMAIN = {Element$_1$, ….., Element$_n$}

Table 5: abstract domain signature

(dynamic)? abstract domain ID_DOMAIN

Table 6: basic domain signature

basic domain ID_DOMAIN

In AsmetaL, a function is considered as an entity that replaces variables in programing languages. As shown in Figure 5, a function could be either a basic function or a derived function. Basic function consists of static function (*cannot be updated during the execution*), and dynamic function (*out function, monitored function, shared function, controlled function, and local function*), as shown in Table 7. Furthermore, dynamic function consists of out function (*responsible of output to environment*), controlled function (*only updated by the machine*), monitored function (*only updated by the environment (user), thus, it cannot appear in the left side in update rule*), and shared function (*updated by machine and environment*)

Derived function (*the value of derived function depends on the input*), as shown in Table 9, where its value depends on the input.

Table 7: static function signature

static ID_FUNCTION : ID_DOMAIN ('->' ID_DOMAIN)?

Table 8: dynamic function signature

local	(dynamic)? local ID_FUNCTION : (ID_DOMAIN '->')? ID_DOMAIN
controlled	(dynamic)? controlled ID_FUNCTION : ID_DOMAIN ('->' ID_DOMAIN)?:
Shared	(dynamic)? shared ID_FUNCTION : ID_DOMAIN ('->' ID_DOMAIN)?
monitored	(dynamic)? monitored ID_FUNCTION ':' ID_DOMAIN ('->' ID_DOMAIN)?
out	(dynamic)? out ID_FUNCTION : ID_DOMAIN ('->' ID_DOMAIN)?

Table 9: derived function signature

derived ID_FUNCTION : ID_DOMAIN ('->' ID_DOMAIN)?

The body section consists of all domains, functions, rules, and invariants definitions respectively. Concrete domains and static functions value is set in the definition statements. A derived function is defined in term of input.

There are two main rule declarations supported by AsmetaL language: i) Turbo rule declaration, which takes a set of parameters and provide an optional return value in which its type is defined in the rule header as shown in Table 11. In addition, they are called using parentheses. ii) Macro rule declaration, which takes a set of parameters, but, do not return any value and are called using squired brackets as shown in Table 10. When decelerating rules the order of declaration matters, in other words, if rule r_a calls r_b, then declaration of r_b must precede the declaration of r_b, thus it is impossible to have recursive call between rules e.g., r_a calls r_b and r_b calls r_a.

Table 10: macro rule declaration

(macro)? rule ID_RULE ((variable in ID_DOMAIN (, variable in ID_DOMAIN)*)?)

Table 11: turbo rule declaration

turbo rule ID_RULE ((variable in ID_DOMAIN (, variable in ID_DOMAIN)*))? (in ID_DOMAIN)? '=' rule

The main rule (Table 12) is the rule that will be executed first when running AsmetaL specification. It is possible not to specify a main rule in case a module is exported. In addition, the initialization section is optional, where the initial states are set. AsmetaL allows only a single default state and multiple of non-default state initialization.

Table 12: main rule declaration

main (macro)? rule ID_RULE ((variable in ID_DOMAIN (, variable in ID_DOMAIN)*))? (in ID_DOMAIN)? '=' rule

AsmetaL supports around 15 type of rules each for a particular purpose as shown in Figure 4. Rules are classified into six classes: i) Basic rule includes skip rule (*does nothing*), macro rule call (*the call of macro rule declaration*), block rule (*executes multiple inner rule in a parallel manner. Note that it must contains at least 2 rules*), conditional rule (*executes branch rules based on guard condition*), choose rule (*provides a non-deterministic behavior by using an arbitrary term form domain that satisfies the guard condition*), forall rule (*executes do-block rule for all term in a domain that satisfies the guard condition*), let rule (*executes in-block rule while assigning terms to variables*), and extend rule (*extends a domain with terms*). ii) Update rule (*updates the value of function. As mention before, the machine cannot update the value of monitored function*). iii) Turbo return rule. iv) Term as rule. v) Derived rule. vi) Turbo rule includes sequence rule (*executes multiple inner rule in a sequential manner. Note that it must contains at least two rules*), iterative rule (*loop through do-block rule*), turbo call rule (*the call for turbo rule declaration*), and turbo local state rule (*internal rule used inside turbo rule to return the local state variable*). Table 13 shows the syntax of each type of rules.

Table 13: AsmetaL rule structures

Skip	skip
Macro rule call	ID_RULE '[' (Term (',' Term)*)? ']'
Block	par Rule (Rule)+ endpar
Conditional	if Term then Rule (else Rule)? endif
Choose	choose VariableTerm in Term (',' VariableTerm in Term)* with Term do Rule (ifnone Rule)?
Forall	forall VariableTerm in Term (',' VariableTerm in Term)* (with Term)? do Rule
Let	let '(' VariableTerm '=' Term (',' VariableTerm '=' Term)* ')' in Rule endlet
Extend	extend ID_DOMAIN with VariableTerm (',' VariableTerm)* do Rule
Update	(LocationTerm | VariableTerm) ':' Term
Turbo return	(LocationTerm | VariableTerm) '<-' TurboCallRule

Term as rule	FunctionTerm | VariableTerm
Derived	whilerec Term do Rule while Term do Rule
Sequence	seq Rule (Rule)+ endseq
Iterative	iterate Rule enditerate
Turbo call	ID_RULE '(' (Term (',' Term)*)? ')'
Turbo local state	(LocalFunction '[' Rule ']')+ Rule

AsmetaL supports multiple initializations including a single optional default initialization. Each initialization provides the initial state of domains, functions, and agents. AsmetaL simulator can handle uninitialized domains, functions, and agents (*default values are set to undef*); however, it is recommended to initialize all predicates.

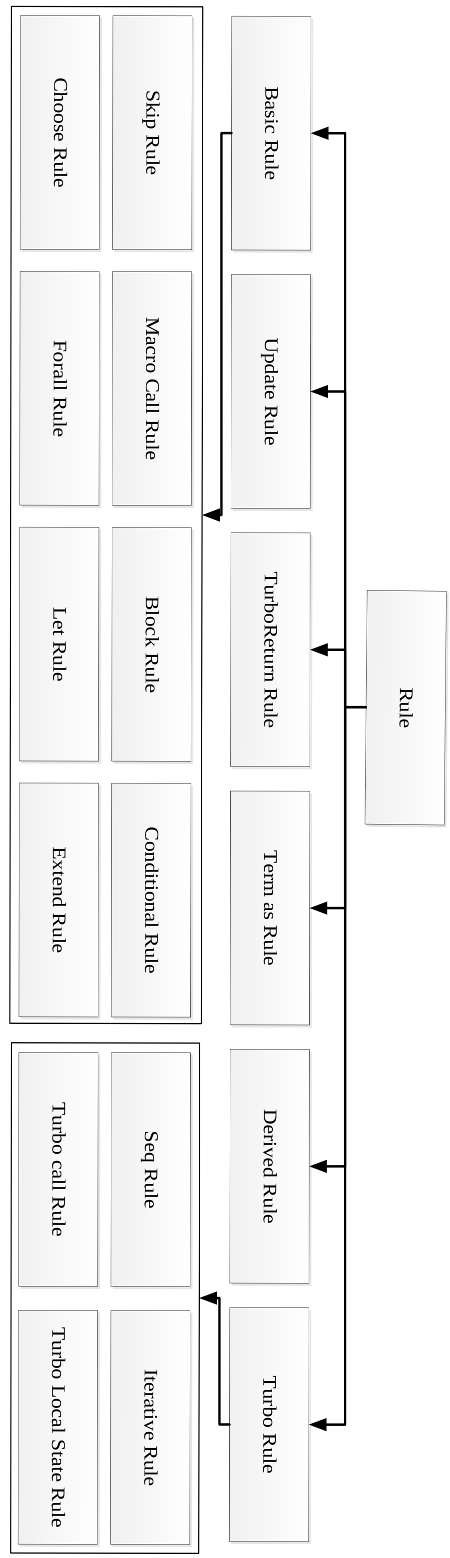


Figure 4: AsmetaL rule types

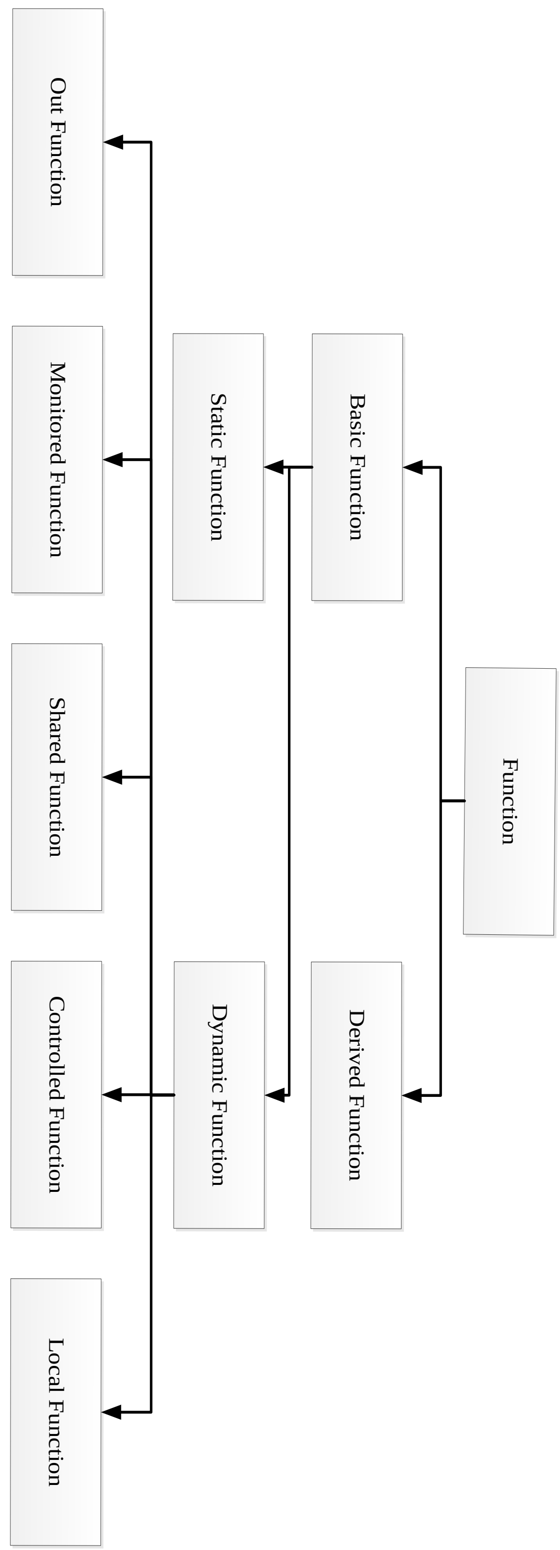


Figure 5: AsmetaL function types

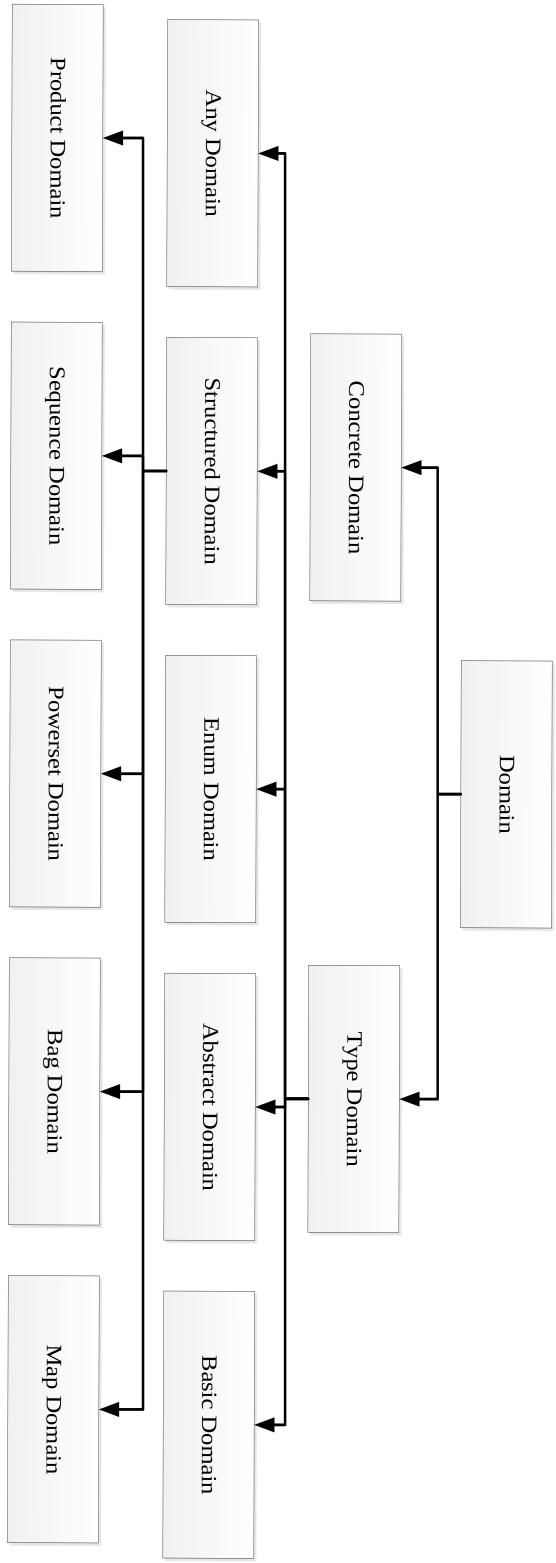


Figure 6: AsmetaL domain types

For further information about the language structure, please refer to the full language grammar (*EBNF grammar*) [23].

2.1.5 AsmetaL Tools

- ASMeta compiler (AsmetaLc) is a text to model complier that parses AsmetaL specification in order to check its consistency with respect to itself. It is available for download via [24].
- ASMeta simulator (AsmetaS) is run-time simulator that executes AsmetaL specification modules in a scenario based. It is available for download via [25].
- ASMeta validator (AsmetaV) is AsmetaL specification validation tool. It is available for download via [26].
- ASMeta modelchecker [27][28](AsmetaSMV). It is available for download via [29].
- ASMEE [30],[31] is an eclipse plugin that add the support of AsmetaL environment for eclipse IDE.
- AsmetaRE [32].
- NuSMV [33].
- NuSeen [34].
- NuSMV model advisor [35].
- ATGT [36] is a test generation tool that support generating test suite for Asmetal modules based on coverage criteria.
- ATGT Boolean [37][38] is a test generation tool that enforce optimization for efficient test suite generation.

- SCA-ASM [39].

2.2 Mutation Testing

Recently according to Jeevarathinam et al.[40], the interest of research has increased in the field of software mutation testing emerged from the importance of software testing process. The demand of higher quality of software products increased the need for better testing methodologies. Software testing process aims to detecting bugs in the system-to-be as well as increasing the confidence of the end user based on many tasks such as unit testing, integration testing, system testing, and specification validation. These tasks share the process of designing the test cases is non–trivial task and considered to be subjective task due to the fact that different outputs is resulted depending on the human factor involved in. Thus, test cases produced by different testers may vary in the level of effectiveness. Although there are some testing techniques such as coverage criteria that aims at increasing the effectiveness of a test suite, however, it does not consider the testing data selection. Hence, there is a subtle need of a systematic methodology to assess the effectiveness of test cases.

Mutation testing, was first introduced in 1971 by Lipton [41], aims to provide a numerical representation of the adequacy of the test cases (*testing suite*). Based on two main hypotheses, **Competent Program Hypothesis** [42] which assumes that developers are smart people and they try to develop system-to-be in such a way that it is close to correct, thus, the typical faults are considered to be miner faults. Based on that hypothesis, it determines how to inspect and test systems in a way that minor faults are more potentially to exist; therefore it should be carefully tested. In addition, complex faults are less likely to exist. Second, **Coupling Effect Hypothesis** [43] which assumes that complex faults are coupled with minor faults considering that complex faults are decomposed of a set of minor faults. In other words, the data selected to detect all miner faults would detect most of the complex faults. Thus, the detection and elimination of minor faults would detect and eliminate complex faults simultaneously.

Figure 7 illustrates the typical procedure to generate mutants. Given a program P with a test suite T and a set of mutant P' that does not include P. The typical procedure to generate mutants starts by running P against T. It is important that P passes without detecting any failure, thus, the fault will not propagate to the generated mutants. The mutants will be generated based on a predefined set of operators, which present systematic rules to generate mutants. For the first order mutants only a single mutation operator must take place. If T able to distinguish P from P', it is considered that all mutants in P' are *killable* and eliminated from any further considerations. While the living (*non-killable*) mutants are either mutants that could be killed but the test suite is not sufficient, Therefore, more test cases must be add to kill these living mutants or equivalent mutants. The equivalent mutants are those mutants that syntactically differ from P but have identical behavior to P. More details about equivalent mutants is presented in section 2.2.2.

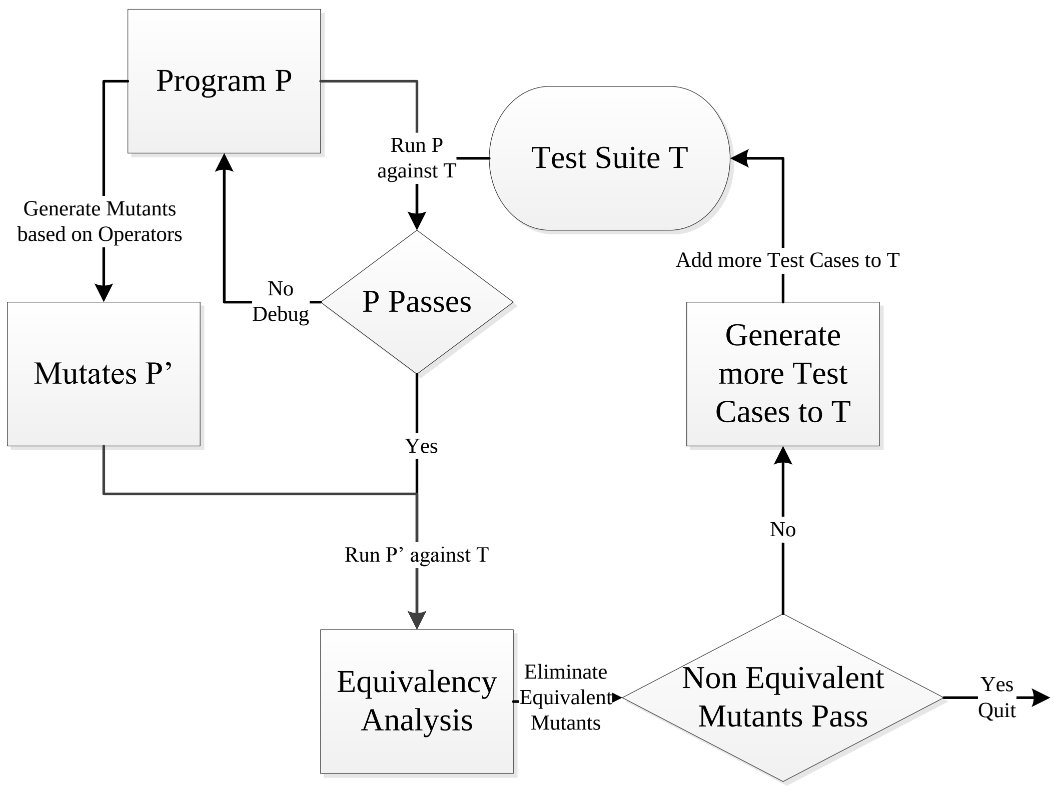


Figure 7: Typical procedure of mutation testing

2.2.1 Mutation Score

Mutation testing does not measure the presence of potential faults in the system-to-be rather than the adequacy of the test suite. The fewer living mutants resulted from preforming mutation testing, the more adequate is the test suite. A mathematical representation of that concept through mutation score denoted by *MS*. Mutation score measures the ratio of the killable mutants denoted by M_K to the all non-equivalent mutants denoted by $M\text{-}M_E$. Equation 1 shows the mathematical formula of the mutation score. The higher is the mutation score, the more adequate the testing suite. It should be noted that mutation score is weighted metric and its value range is [0, 1].

$$MS(P,T) = \frac{|M_k|}{|M - M_E|}$$

2.2.2 Equivalency Analysis Techniques

An equivalent mutant M_E is a syntactically different from the original program P; however, it has the identical behavior of the original program. Thus, no test case exists that can distinguish the output/behavior of the original program P from the equivalent mutant M_E. In order to obtain an accurate mutation score that reflects the adequacy of testing suite, equivalent mutant must be eliminated from further process once they have been detected. According to Jia et al.[44], the problem of detecting equivalent mutations is generally undecidable problem. It could result from many scenarios such as dead code, non-propagated fault and un-triggered events. Typically, they are detected manually in which it requires a lot of time and effort. Many approaches in the literature have been proposed to address the problem of detecting equivalent mutants.

Compiler Optimization Technique, proposed by Baldwin et al.[45], relies on the fact that compliers within the process of compiling the code tend to optimize it, as consequent, many equivalent mutants are generated from the optimization process. By intercepting the optimization process, the number of equivalent mutants will decrease. Offutt et al.[46] proposed Constraint Test Data Generation in which the propagation of fault from input to output of the mutated path is analyzed based on constraints. If the constraints could not be realized then the tested mutant is considered as an equivalent mutant.

Program Slicing Technique [47] based on the conventional procedure, however, it reduces the effort required by adopting the idea of slicing the code so that it is easier to analyze manually. Syntactic Difference [48] considers the idea of different programs consume different resources and have different execution time. Based on these aspects, it could be possible to differentiate between the original program P and mutants.

Different Program Behavior [49] distinguishes the original program P from mutants based on behavior of the interaction between the program/mutants and its external environment rather than output.

2.2.3 Reduction Techniques

Mutation testing generally is considered to be a computationally expensive task. Hundreds if not thousands mutations are generated from the original program P. The most expensive step is the execution of each mutant against the test suite T. Many techniques have been proposed to reduce of the mutation computational cost. Jia [44] classifies them into classes:

2.2.3.1 Cost Reduction Techniques

The cost reduction techniques reduce the number of mutants that must be tested, however, the number of the generated mutant remain identical of the typical procedure.

Acree [50] suggested in his PhD dissertation a novel approach called mutation sampling technique, which basically runs a random set (*x%*) of the entire possible mutants against the testing suite. The procedure can be summarized as follows:

1. List all possible mutants.
2. Randomly select a set of mutants x% of the entire mutants set.
3. Mutation testing is performed on all mutants in the randomly selected set.
4. The remaining mutants are discarded.

Wong and Mathur [51] conducted a study to examine the effectiveness of the sampling technique, they suggested that preforming a mutant sampling on rate of 10% is less effective than the full mutants testing by 16%.

Hussain [52] proposed in his master's thesis a novel approach called mutation clustering by selecting mutants based on clustering algorithm (*K-means and Agglomerative clustering algorithms [53]*) instead of selecting the mutants randomly.

1. List all possible mutants.
2. Apply the clustering algorithm to classify mutants.
3. Select few mutants from each class.
4. Mutation testing is performed on the selected mutants.
5. The remaining mutants are discarded.

Comparing this approach with the previous one, mutant clustering resulted in a reasonable mutation score, while selecting fewer mutants.

Unlike the previous approaches, selective mutation approach reduces the number of mutants by reducing the set of mutation operators to generate fewer mutants.

Many of the proposed techniques are based on N-selective mutation, such as 2-Selective was proposed by Mathur [54], in which eliminates two operators ASR (*array reference for scalar variable replacement*) and SVR (*scalar variable replacement*), the number of mutants will be decrease significantly. This approach maintains a mutation score of 99.99% while the number mutant is decreased by 24%.

In addition, 4-Selective was proposed by Offutt [55], in which eliminates four operators, the number of mutants will be decrease significantly. This approach maintains a mutation score of 99.84% while the number mutant is decreased by 41%.

The 6-Selective was proposed by Offutt [56], in which eliminates six operators, the number of mutants will be decrease significantly. This approach maintains a mutation score of 88.71% while the number mutant is decreased by 60%.

Wong and Mathur [54] proposed constraints approach in which mutant is generated based on ABS (*absolute value insertion*) and ROR (*relational operator replacement*) operator.

Since, ABS mutants are killed using test cases cover input domain partitions and ROR mutants are killed using test cases generated based on the mutant predicate.

Jia and Harman [57] introduced a new approach to mutation testing, in which it finds higher order mutants that are rare, valuable and harder to kill. Considering single operator mutant is a first order mutant, the higher order mutant is produced by replacing multiple first order mutants. As a result, fewer higher order mutants that cover all first order mutants result in a same mutation score.

Polo et al.[58] proposed an improved algorithm to generate second order mutant for the first order mutant. Their experiment demonstrates that their approach reduces the cost by 50% while achieving the similar effectiveness test.

2.2.3.2 Execution Cost Reduction Techniques

This class of mutation reduction focuses on the improving the test execution process to reduce the cost of mutation testing.

Strong mutation [43] testing is referred to the process where a mutant is killable if the final result of the execution defers than the expected final result of the original program.

Weak mutation [59] testing is referred to the process where a mutant is killable if the intermediate (state after the execution of the mutant instruction) result defers than the intermediate of the original program. Weak mutation testing trades of the cost of execution and the effectiveness of mutation testing reduces the effort of fully execution of the program, but it reduces the effectiveness of the mutation testing.

Firm mutation [60] testing is referred to the process where a mutant is killable if the continues intermediate possibilities in which it combine the strong and weak approaches.

2.2.3.3 Runtime Optimization Techniques

Interpreter based technique [61] is basically any mutant is generated from the source code directly.

$$Mutation\ Cost\ = \sum interpretation\ cost$$

The interpreter based technique provides flexibility and efficiency form small programs.

Compiler based technique [62] is basically any mutant is compiled to binary code and then it is executed, since the execution of binary code is much faster than the interpreter.

$$Mutation\ Cost\ = \sum (compilation\ cost\ +\ execution\ cost)$$

Mutant schema technique [63] is basically for all mutants a single super mutant is created and complied once with a meat program for each individual original mutant.

$$Mutation\ Cost\ =\ single\ compilation\ cost\ +\ \sum execution\ cost$$

Bytecode translation technique [64] is basically all mutants are derived from the original compiled program without the need of any compilation cost of any mutant. This technique support applying mutation testing without the need of the source code of the program tested. However, it is subjective to the nature of the language itself.

$$Mutation\ Cost\ =\ \sum execution\ cost$$

Aspect oriented mutation [65] is basically preforming mutation testing on the fly, by applying two iterations:

1. Get the result of the original program.
2. Generate and execute the mutants.

There are other approaches that focus on reducing the execution cost of mutation testing based on distributed systems and parallel mutation testing, however, it is not part of scope in that research.

2.3 Chapter Summary

In this chapter, we have provided a general definition and basic notation for Abstract State Machines. In addition, several ASM languages and environments were briefly reviewed and compared based on simple comparison criteria. Since our intention is to propose a mutation approach for AsmetaL language, we have provided an in-depth review for the structure of AsmetaL. The second part of that chapter provides a general notion and definition for mutation testing technique, in addition to a review of equivalency analysis techniques.

CHAPTER 3

Testing Abstract State Machines: State of the Art

Many techniques have been proposed in the area of Abstract State Machines testing. In this chapter, we classified ASM-based techniques into five main categories. i) FSM generation from ASM techniques, which uses FSM well, defined testing techniques to test ASM. ii) Conformance testing technique to assure that the implementation is corresponding to the specification. iii) Coverage criteria for test case generation for ASM. iv) Model checking technique to ensure the consistency between implementation and specifications. v) Test generation technique based on the aforementioned techniques. In addition, we have spotted some works been done in the area of formal specification testing such as FSM, State charts etc.

3.1 Testing Abstract State Machines

3.1.1 Generation of Finite State Machines (FSM) from ASM

Finite State Machines (FSM) is a computational model that consists of states, transitions, input/output. According to Belinfante et al.[66], an ASM can be considered as a generalization of an FSM. The main difference mentioned in the literature is that ASM could have finite or infinite number of states, while FSM must have finite number of states. In many approaches such as ASM testing and ASM test case generation, ASM model is transferred into FSM to take advantage of the well-defined analysis techniques [67]. In addition, ASMs tend to have more states compared to FSMs. Unfortunately; the transformation process of an ASM to a FSM preserves some of the properties of the ASM model but not all of them.

State Exploration

Barnett et al.[68] have proposed an approach that is similar to the fundamentals that model checker operates on. It is so-called state space exploring, since it starts from the initial state of the ASM model and then explore the next states. Unfortunately, the exploring process suffers from the state explosion problem, where the exploring step tries to cover all possible next states and end up with infinite possibilities. Thus, the exploring step must be subject to prune techniques in order to make the space of exploration manageable. Mostly three pruning techniques are used: i) State abstraction where each state in the FSM model (concrete state) is mapped to a state in the ASM model (abstract state); the breakpoint is when next state is already mapped. ii) Filters techniques are based on removing all states that do not comply with certain domain-based conditions before being explored. iii) Model coverage technique defines the amount of coverage that must be achieved in order to stop exploring. The transformation process starts by generating domain specific parameters, which are based on ADF (*access driven filter*). These parameters are used to identify abstraction properties that rule the prune process. The abstraction properties identification is manual task and subjective to the experience.

Belinfante et al. [66] proposed another technique for reducing the number of states in the resulted finite state machine. This technique take advantage of the guard condition; if there is an existence of two test cases where one of them results in a true value for the guard condition and the other one resulted in false, then that guard condition called distinguishable condition. On the other hand, if they do not exist, then the two adjacent states, which have the update condition between them, are called equivalent states. By merging the adjacent equivalent states into one state called hyper-state, the number of state is reduced to finite number. In addition, DNF is another approach, which attempts to investigate each clause of the guarded condition.

3.1.2 Conformance Testing

Conformance testing is one of the important types of software testing, where the objective of that type of testing is the assurance that the implementation is corresponding to the specification. As mentioned in section 2.1.2, ASMs are executable [69], thus, conformance testing can be used to validate the conformance of implementations to the specifications. According to Grieskamp et al.[70], conformance testing is carried out as shown in Figure 8 : i) the inputs for conformance testing are specification and implementation. ii) The output is that the implementation either conforms or does not conform to specification. Originally, the specification is used to derive test cases and the expected behavior. Whenever the implementation is completed, these test cases are run against the implementation. The conformance of the expected and actual behavior determines if the implementation is conformed to the specification. However, ASMs could have infinitely many states, where it is impossible to apply the original conformance testing. Thus, conformance testing must be modified to accommodate this dilemma. Generally, there are two approaches to preform ASM conformance testing. i) Labeled Transition Systems (LTS) – based, and ii) Finite State Machines (FSM) – based.

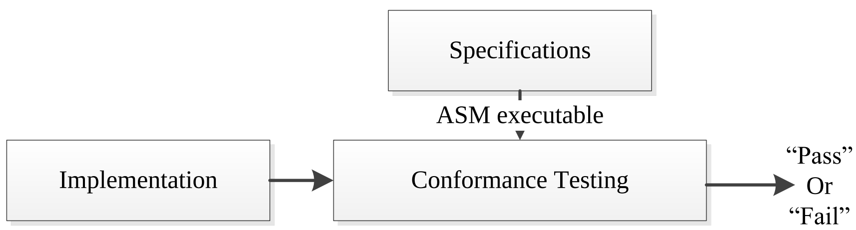


Figure 8: Conformance Testing Concepts

LTS – based

"A labeled transition system is a structure consisting of states with transitions, labeled with actions, between them" [70]. Labeled transition systems [71] based testing is one of the testing techniques that the conformance testing could be carried out. It can be applied to any input – output transition based system. Compared with the FSM based conformance testing, it is a general testing approach based on the model specification. In addition, LTSs main characteristic is that it does not depend on transforming the ASM model to FSM Model in order to perform testing. Thus, the overhead of transformation is eliminated. Unlike, the FSM conformance testing which, makes distinguish between input and output of the interaction. In order for a transition to be carried out, all participating processes must have a transition at the current state that results in the next state. The Interactions in LTS is considered as inputs to the FSM where the outputs of FSM cannot be mapped in to LTS.

According to Grieskamp et al.[70], LTS normally captures the external behaviors of the system with it environment, thus, it is a black box testing where it validate the conformance of implemented model of the system to the specification model of the system. In addition to the deterministic and sequential interactions, LTS supports both nondeterministic (by introducing the refusal set which identified by the blocking behavior) and parallel interactions. LTS relies on what are so-called conformance relations (interactions of interest). Many researches have been conducted to generate test suite for LTS model by deriving FSM model, however, the size transitions of FSM model is huge compared with LTS model.

FSM – based

Many well-defined techniques to preform conformance testing for the FSM have been proposed in the literature. These techniques are not specific to FSM generated from ASM; they are general techniques that are applicable to any FSM model. Examples of such techniques include D-method [72], W-method [73], U-method [74] and Uv-method [74]. The review of these techniques is out of the scope of this research.

3.1.3 Coverage Criteria

In typical software testing, the coverage criteria determine the testing requirements that achieve full coverage where minimal test cases are generated to fulfill the testing requirements. Unfortunately, it is considered as costly and inconvenient. However, specification based testing reduces the cost, since, ASMs are executable in its nature as mentioned in section 2.1.2, they can be automated to contribute to the testing process and reduce the testing cost. ASMs specifications are used to automate the generation of the test oracle (expected output), assessment of the adequacy of test suite and generation of testing sequence. In order to get the maximum benefit of the testing coverage criteria, it is important to get an overview of the existing coverage criteria that are tailored for ASMs specifications. Gargantini and Riccobene [75] proposed a classification (from the weakest to the strongest) of coverage criteria:

- State Coverage (node coverage): for every state in ASM model, there must be at least one testing sequence in which a state is exercised |S|.
- Rule Coverage: for every rule in ASM model, there must be at least one testing sequence in which the rule is fired.
- Rule Update Coverage: for each rule update for all rules in ASM model, there must be at least one testing sequence in which the rule update is fired and the rule update is not trivial.
- Parallel Rule Coverage: for every n-tuple of rules, it must be either unfirable or there must be at least one testing sequence that fires all n rules simultaneously.

- Strong Parallel Rule Coverage: for every k-tuple of rules, it must be either unfirable or there must be at least one testing sequence that fires all k rules simultaneously. Where k is $1<=k<=n$.
- Modified Condition Decision Coverage: for each clause C_i of guard condition, there must be testing sequences in which C_i is once true and once false, where other clauses are fixed and the guard condition is affected(once true and once false).
- Multiple Condition Coverage: for each and every clause of every guard condition, there must be testing sequences in which all combination of clauses is explored 2n.

3.1.4 Model Checking

The basic concept of model checker as described by Clarke et al.[76], is to ensure the consistency between implementation and specifications by providing a proof for a certain property of a model that is true in any possible state of the model. Originally, the model is a finite state model that will be transferred into a Kripke structure, while the specifications are a temporal logic expression in the form of either linear expression or branching expression. The output of the model checker is one of the following cases: i) Return true, that means the property holds for all possible state identified by the temporal logic expression. ii) Return false, with counter example in which a state violates the temporal logic expression for that property. iii) No conclusion, in some cases the model checker suffers from state explosion problem in which it will try to cover all possible execution and ends up with infinite number of possibilities that will consume all of the available resources.

The model checking technique [77] is considered to be computationally expensive due to the state exploring process that may lead to infinite possibilities (state explosion). Thus, it does not support ASMs specification natively, since ASM is infinite in its nature. Many works have been done to extract FSM models from ASM (see section 3.1.1) to take advantage of the existing techniques provided by model checking.

3.1.5 ASMs Test Case Generation

3.1.5.1 FSM-based

This approach [68] is based on AsmL which supports generation of FSMs from ASM specification as discussed in section 3.1.1. The process of generating a test suite implies traverse all the states of FSM starting by the initial state and ending by the same initial node based on Chinese postman tour algorithm. Unfortunately, the resulted test suite only archives node based coverage, which is considered as a weak coverage criteria. Grieskamp et al. [78] discussed another approach based on FSM which generates test cases using a graph reachability algorithm to explore nondeterministic FSM state space controlled by the original AsmL meta-programming. This technique implies a depth-first search algorithm starting at the initial state.

3.1.5.2 Model Checking-based (for coverage criteria)

For coverage criteria where the testing requirements are defined, a model checking based technique (see section 3.1.3) can be used. Model checking is a widely used technique in the FSM realm in which it shows whether a certain properties can hold in all possible states. Generally, a model checker takes a model and a specification as input, and examines all possibility based on state explosion mechanism [97]. The idea of using model checker lies in the fact that model checker provides a counter example [75]. However, model checking based technique is considered to be computationally expensive. Moreover, model checking operates on finite space domain, while, ASMs specification could be infinite in domain space [77].

3.2 Mutation Testing of Formal Specifications

Although mutation testing has mostly been applied at the source code level, it has also been applied to formal specifications [44]. Fabbri et al.[79] have applied specification mutation to validate specifications based on Finite State Machines (FSM). They have proposed 9 mutation operators, representing faults related to the states (*e.g.,, wrong-starting-state, state-extra, etc.*), transitions (*e.g.,, event-missing, event-exchanged, etc.*) and outputs (*e.g.,, output-missing, output-exchanged, etc.*) of an FSM. Fabbri et al.[80] have defined mutation operators for Statecharts, an extension of FSM formalism, while Batth et al.[81] have applied mutation testing to Extended Finite State Machines (EFSM) formalism. In the ASM context, Hassine [82], [83] has defined a set of generic mutation operators for Abstract State Machines. The proposed operators have been classified into three main generic classes: (1) ASM domain mutation operators, (2) ASM function update mutation operators, and (3) ASM transition rules mutation operators. In this work, we refine the ASM-based operators introduced in [83] to accommodate the AsmetaL language.

Hierons and Merayo [84] have investigated the application of mutation testing to Probabilistic (PFSMs) or stochastic time (PSFSMs) Finite State Machines. The authors have defined new mutation operators representing FSM faults related to altering probabilities (PFSMs) or changing its associated random variables (PSFSMs) (i.e., the time consumed between the input being applied and the output being received).

Formal specification languages to which mutation testing has been applied include Finite State Machines [79],[84], and [85], Statecharts [80], Petri Nets [86], and Estelle [87].

Fabbri et al.[79] have applied specification mutation to validate specifications based on Finite State Machines (FSM). They have proposed 9 mutation operators, representing faults related to the states (e.g.,, wrong-starting-state, state-extra, etc.), transitions (e.g.,, event-missing, event-exchanged, etc.) and outputs (e.g.,, output-missing, output-exchanged, etc.) of an FSM. In a related work, Fabbri et al.[80] have defined mutation operators for Statecharts, an extension of FSM formalism, while Batth et al.[81] have applied mutation testing to Extended Finite State Machines (EFSM) formalism.

3.3 Chapter Summary

In this chapter, we presented, in the first part, Abstract State Machines testing state of the art including the generation of FSM from ASM specifications, ASM conformance testing, test case generation coverage criteria, ASM model checking. In addition, we reviewed

FSM-based, and model checking based test case generation techniques. In the second part, we spot some of the formal specification mutation testing works.

CHAPTER 4

AsmetaL Mutation Testing Approach

In this chapter, we present our AsmetaL mutation testing approach. We describe the proposed mutation testing methodology, and the proposed set of mutation operators for the AsmetaL language. In addition, we evaluate our set of operators experimentally using a set of case studies of different sizes.

4.1 AsmetaL Mutation Testing Approach

Figure 9 illustrates our AsmetaL mutation testing approach. Six main tasks were conducted:

Task 1: Generate initial test suite T for AsmetaL specification P using ATGT tool (*A test generation tool*) and set a mutation score threshold.

Task 2: Run T against P to detect any fault, thus, assure elimination of any propagated fault to the generated mutants.

Task 3: Generate mutants P' (*automated*) from P based on the proposed mutation operators.

Task 4: Run the initial test suite against P' (*automated*). All killed mutants will be discarded from any further processing, therefore, only live mutants will be considered for the next steps.

Task 5: Perform equivalency analysis (*manual*) on live mutants, in order to eliminate equivalent mutants from any further processing.

Task 6: Generate more test cases and add them to T in order to kill living non-equivalent mutants.

Task 7: Run generated test cases against P'.

Repeat Steps 6 and 7 until mutation score threshold is achieved.

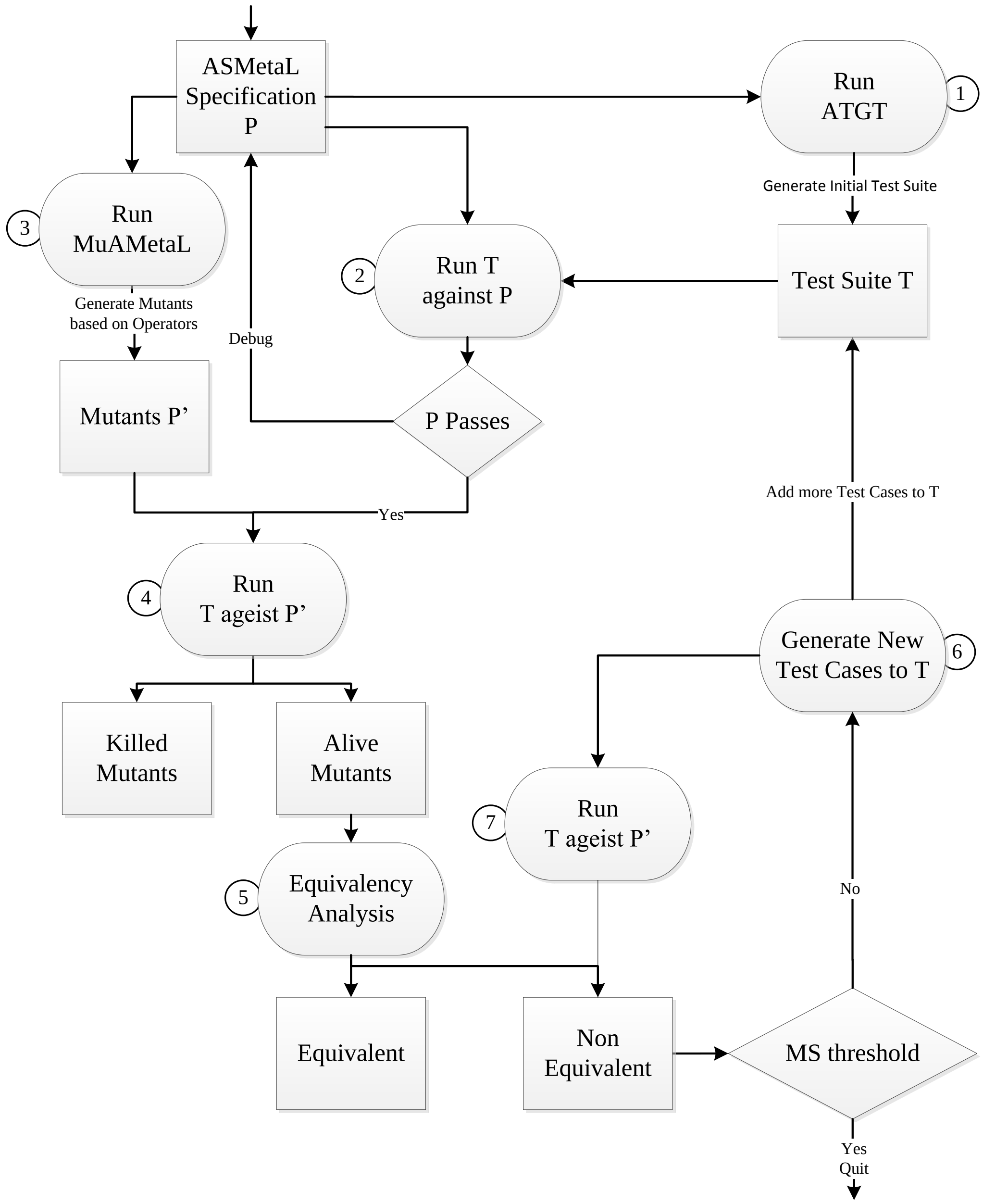


Figure 9: AsmetaL mutation testing procedure

4.1.1 Design of mutation operators

Our designed AsmetaL-based mutation operators will follow the principles provided by [95] in which only first order mutation testing will be considered and all of the generated mutants are syntactically correct. In addition, mutation operators will address potential faults.

This phase includes the following tasks:

1. Investigate ASM fault classes.
2. Design mutation operators based on AsmetaL syntax's.
3. Investigate the validity of each operator.

Assumptions:

- This study considers first order mutants only.
- This study considers mutation operators that produce syntactically correct mutants.
- Only potential faults resulting from (the defined classes of faults) will be addressed by this study.

The proposed approach relies on a three steps generation process as shown in Figure 10.

Step 1: Create Mutant M.

Step 2: Validate syntax of M using AsmetaLc.

Step 3: Check S ≠ M syntactically. Where S is the original specification.

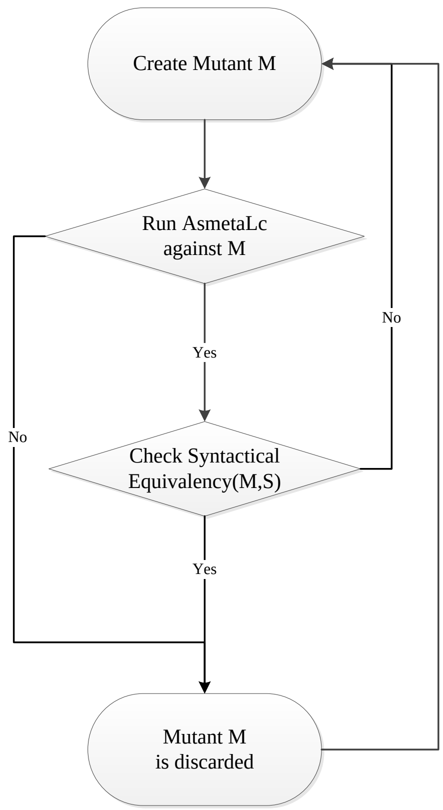


Figure 10: Mutant generation process

4.1.2 AsmetaL mutation tool design and implementation

MuAsmetaL is name of prototype tool that will be developed during this research. It will be both command line and GUI java based tool that will give the user the ability to view/edit AsmetaL specifications, parse specifications, run the specifications, and generate mutants and execute them.

4.1.3 Empirical evaluation of the proposed approach

We intend to validate theoretically the research hypothesis by developing the proposed approach. Different theories and techniques are involved in the support of the proposed verification cycles of Figure 9. Some of them, such as Mutation testing, equivalency analysis, and test case generation already exist. Others are still to be developed as part of this research:

- Design of the AsmetaL mutation operators.
- Apply the proposed operators (automatically).
- Design of test oracle for AsmetaL (verdict on passing/failing test cases).
- Implementation of the MuAsmetaL Tool (CLI and GUI).

We intend to validate our approach through its application to a wide range of AsmetaL specifications.

4.1.4 Selective mutation

Mutation testing is known for its high computational cost. In order to reduce the computation cost a selective set of mutants will be chosen based on two criteria: i) level of effectiveness achieved. ii) Reduction of computation cost.

The empirical data collected from empirical evaluation is used to assess the effectiveness of applying selective-based and random mutation testing. This would allow for computation cost reduction, without affecting the effectiveness of the proposed methodology.

4.2 AsmetaL Mutation Operators

Mutation operator is a rule in which it governs the way fault is injected into the original specification to produces mutants. Typically, each operator tends to cover a real potential fault that might exist in the original specification. In order to generate mutants, we have to define each mutation operator. The defined set operators must provide a complete coverage all of the aspects of AsmetaL grammar and to including all of the language constructs. We have classified AsmetaL mutation operators into 5 different classes as follows:

4.2.1 Function mutation operators

AsmetaL functions are classified into static (*not updated at run-time*), derived (*its return value is subjected to its inputs*), and dynamic. Dynamic functions are further classified as monitored, controlled, shared, out, and local. Local dynamic are declared and used only in the scope of a turbo transition rule with local state. An AsmetaL function can be mutated using:

- **Function Type Permutation Operator** (FTP) (Table 14). FTP operator replaces a dynamic function type with other types (e.g.,, ***controlled***, ***monitored***, ***shared***, ***out***). It is worth noting that if a controlled/shared/out function appears in the left hand side of an update rule, then mutating the function type to ***monitored*** would produce an invalid mutant. Mutate function types from static/derived to dynamic and vice versa would produce invalid mutants.

Table 14: FTP operator example

Operator	Original AsmetaL code	Mutant AsmetaL code
FTP	controlled value : Integer	monitored value: Integer

4.2.2 Rule mutation operators

We define 28 rule-based mutation operators for the AsmetaL language (Table **15**):

- **Rule Guard Condition Replacement Operator** (RGCR): Replaces a guard condition with another existing guard condition. The application of the operator may result into invalid mutants in case the new guard has undefined variables in the current scope.
- **Then Rule Replacement Operator** (TRR): Replaces then rule with an existing rule (*except variable and function terms*).
- **Else Rule Replacement Operator** (ERR): Replaces the else rule with an existing rule (*except variable and function terms*).
- **Main Rule Replacement Operator** (MRR): Replaces the main rule declaration with an existing macro rule declaration.
- **Parallel Block to Sequence Operator** (PB2S): Converts a ***block*** rule to a ***sequence*** rule.
- **Sequence to Parallel Block Rule Operator** (S2PB): Converts a ***sequence*** rule to a ***block*** rule. S2PB operator may lead to inconsistent updates. It is worth noting that the parser can discover only trivial inconsistent updates (for example a function whose value is modified by two parallel instructions in the same rule). The other inconsistent updates will occur at run-time.
- **Add Rule Operator** (ARO): Adds an existing rule to a ***block*** rule or to a ***sequence*** rule.
- **Replace Rule Operator** (RRO): Replaces a rule within a ***block*** or a ***sequence*** rule with an existing rule (except variable and function terms).

- **Sequence Block Statement Deletion Operator** (SBSDL): Removes a single rule from a ***block*** or a ***sequence*** rule. At least three rules should exist in the ***block***/***sequence*** rule.
- **Sequence Rule Order Permutation Operator** (SSM): Exchanges the order of a pair of rules in a ***sequence*** rule.
- **Choose DoRule Replacement Operator** (CDoR): Replaces the rule defined in a choose rule with an existing rule having the same type.
- **Choose IfNoneRule Replacement Operator** (CIR): Replaces ***ifnone*** rule in a ***choose*** rule with an existing rule having the same type.
- **Choose Rule Exchange Operator** (CRE): Exchanges the do rule with the ***ifnone*** rule. In case ***ifnone*** rule is not defined, the do rule is duplicated to serve as the ***ifnone***. Applying CRE may produce invalid mutants in case the chosen variable does not exist within the scope of the do block.
- **Choose Domain Replacement Operator** (CDR): Replaces one domain of the ***choose*** rule by a compatible one (e.g.,, different integer sub-domain).
- **Forall DoRule Replacement Operator** (FDoR): Replaces the do block defined in a ***forall*** rule.
- **Forall Choose Rules Permutation Operator** (FCRP): Replaces ***forall*** rule with a ***choose*** rule and vice versa. The difference between both types of rules is that:
    - **Choose rule** assigns to each variable an arbitrary value from domain that satisfies the guard condition in order to substitute it in the do block.
    - **Forall rule** assigns to each variable all values from domain that satisfies the guard condition in order to substitute it in the do block.

- **Rule to Skip Rule Operator** (RTS): replaces a non-skip rule with the skip rule.
- **Stuck Switch to Specific Case Operator** (SSSC): Mutate the selector of a switch case rule to force the execution of a specific case.
- **Switch Case Permutation Operator** (SCP): Exchanges a pair of switch case rules in ***case rule***.
- **Case Rule Replacement Operator** (CRRO): Replaces the selected rule to be executed as part of a case selection by another existing rule.
- **Delete Switch Case Operator** (DSC): Deletes a single case from a ***case rule.***
- **Let Rule Variable Assignment Operator** (LRVA): Assigns a different term to a variable within a let rule.
- **Let Rule Replacement Operator** (LRR): Replaces the in-block rule by any existing rule.
- **Let Rule Variable Replacement Operator** (LRVR): Replaces a variable within a let rule by an existing variable term.
- **Extend Domain Replacement Operator** (EDR): Replaces a domain of the ***extend*** rule by a compatible one (e.g.,, different abstract domain).
- **Extend Rule Replacement Operator** (ERRO): Replaces do block by any existing rule in extend rule.
- **Extend ID Replacement Operator** (EIR): Replaces variable, in which domain is add universe of domain, by any existing rule.

Declarations needed for the examples:
Signature: *domain Interval subsetof Integer* *domain IntervalB subsetof Integer* *dynamic abstract domain Products* *dynamic abstract domain Person* *Definitions:* *domain Interval= {1..10}* *domain IntervalB= {1..11}*

Table 15: turbo rule operator's examples

Operator	Original AsmetaL code	Mutant AsmetaL code
RGCR	choose $v in Interval with $v>10 do r_rule[$v]	choose $v in Interval with $v=10 do r_rule[$v]
TRR	if $a=10 then r_ruleA[]	if $a=10 then r_ruleA[]
ERR	if value=10 then r_ruleA[] else r_ruleB[] endif	if value=10 then r_ruleA[] else r_ruleC[] endif
MRR	main rule r_main = r_travel[]	main rule r_main = r_rule[]
PB2S	par r_ruleA[] r_ruleB[] endpar	seq r_ruleA[] r_ruleB[] endseq
S2PB	seq r_ruleA[] r_ruleB[] endseq	par r_ruleA[] r_ruleB[] endpar

ARO	par r_ruleA[] r_ruleB[] endpar	par r_ruleA[] r_ruleB[] r_ruleC[] endpar
RRO	seq r_ruleA[] r_ruleB[] endseq	seq r_ruleC[] r_ruleB[] endseq
SBSDL	par r_ruleA[] r_ruleB[] r_ruleC[] endpar	par r_ruleA[] r_ruleB[] endpar
SSM	seq r_ruleA[] r_ruleB[] endseq	seq r_ruleB[] r_ruleA[] endseq
CDoR	choose $v in Interval with r_ruleA[]	choose $v in Interval with r_ruleB[]
CIR	choose $v in Interval with $v>10 do r_ruleA[] ifnone r_ruleB[]	choose $v in Interval with $v>10 do r_ruleA[] ifnone r_ruleC[]
CRE	choose $v in Interval with $v>10 do r_ruleA[] ifnone r_ruleB[]	choose $v in Interval with $v>10 do r_ruleB[] ifnone r_ruleA[]
CDR	choose $v in Interval with $v>0 do r_ruleA[$v]	choose $v in IntervalB with $v>0 do r_ruleA[$v]
FDoR	forall $v in Interval with $v>10 do r_ruleA[]	forall $v in Interval with $v>10 do r_ruleC[]
FCRP	forall $v in Interval with $v>10 do r_rule	choose $v in Interval with $v>10 do r_rule
RTS	seq r_ruleA[] r_ruleB[] endseq	seq r_ruleA[] skip endseq

SSSC	switch($a) case 1: r_ruleA[] case 2: r_ruleB[] endswitch	switch(1) case 1: r_ruleA[] case 2: r_ruleB[] endswitch
SCP	switch($a) case 1: r_ruleA[] case 2: r_ruleB[] endswitch	switch($a) case 1: r_ruleB[] case 2: r_ruleA[] endswitch

CRRO	switch($c)
	
	    case 1 : r_ruleA[]
	
	    case 2 : r_ruleB[]
	
	endswitch
		switch($c)
		
		    case 1 : r_ruleC[]
		
		    case 2 : r_ruleB[]
		
		endswitch

DSC	switch($a) case 1: r_ruleA[] case 2: r_ruleB[] case 3: r_ruleC[] endswitch	switch($a) case 2: r_ruleB[] case 3: r_ruleC[] endswitch

LRVA	let ($value = 5) in r_ruleA[$value] endlet	let ($value = 10) in r_ruleA[$value] endlet
LRR	let ($value = 5) in r_ruleA[$value] endlet	let ($value = 5) in r_ruleB[$value] endlet
LRVR	let ($value= 5) in r_ruleA[] endlet	let ($x= 5) in r_ruleA[] endlet
EDR	extend Products with $p do value:=$p	extend Person with $p do value:=$p
ERRO	extend Products with $p do value:=$p	extend Products with $p do r_ruleA[]
EIR	extend Products with $p do r_ruleA[]	extend Products with $c do r_ruleA[]

4.2.3 Term mutation operators

Depending on the type of operands, traditional operators (Table **16**) [79] such as Arithmetic Operator Replacement (AOR), Logical Operator Replacement (LOR), Relational Operator Replacement (ROR), and Unary Operator Insertion (UOI) can be applied (Table 4):

- **Arithmetic Operator Replacement** (AOR): Replaces arithmetic operators with other types (e.g.,, +, -, *, /).
- **Unary Operator Insertion** (UOI): Inserts unary operators (+, -), in integer term, real term, natural term, complex term, in addition to function calls returning the following types: Integer, Real, Natural, Complex.
- **Logical Operator Replacement** (LOR): Replaces logical operators with other types (e.g.,, *and, or, xor, implies, iff*).
- **Relational Operator Replacement** (ROR): For basic types, it replaces the relational operator = by ! = and vice versa. For *Integer*, *Real, Natural*, and Char domains, it replaces any relational operator with other types (e.g.,, <, <=, >, >=, =, ! =).
- **Expression Negation Fault** (ENF): Applies negation to guard conditions enclosed within: ***conditional*** term guards, ***exist*** term guards, ***forall*** term guards, ***choose*** rule guards, etc.
- **Literal Negation Fault** (LNF)**:** Applies negation to single Boolean term or function term with Boolean return type.
- **Stuck at True False** (STF): Replace guard conditions by true and false.
- **Absolute Value Operator** (ABS): Inserts the absolute value function to Integer and Real Terms functions return type and constants. The application of this operator

may results in equivalent mutant, in case of applying it to a positive constant, variable, or function.

Table 16: Traditional rule mutation operators examples

• **Operator**	**Original AsmetaL code**	**Mutant AsmetaL code**
AOR	value := $a + $b	value := $a - $b
UOI	value := $a * $b	value := $a * -$b
LOR	if ($a and $b)	if ($a or $b)
ROR	if ($a < $b)	if ($a > $b)
ENF	if ($a and $b)	if not($a and $b)
LNF	if(valid and correct)	if(not valid and correct)
STF	if ($a and $b)	if (true)
ABS	hours := (hours+ 1) mod 3	hours := (abs(hours)+ 1) mod 3

In addition, we have defined the following operators (Table **17**) for AsmetaL terms:

- **Finite Quantification Terms Permutation** (FQTP): Replaces finite quantification terms (***exit***, ***exist unique***, ***forall*** term) with other types. It is worth mentioning that the difference between the three kinds lies in:
    - ***exist*** term returns true if at least single term exists, that satisfies the guard condition. Otherwise, it returns false.

    - ***exist unique*** term returns true if there is only a single term exists that satisfies the guard condition. Otherwise, it returns false.
    - ***forall*** term returns true if there all terms satisfy the guard condition. Otherwise, it returns false.
  - **Term Guard Condition Replacement Operator** (TGCR): Replaces a guard condition with another existing guard condition. The application of the operator may result into invalid mutants in case the new guard has undefined variables in the current scope.
  - **Then Term Replacement Operator** (TTR): Replaces ***then*** term with any existing term.
  - **Else Term Replacement Operator** (ETR): Replaces ***else*** term by any existing term.
  - **Finite Quantification Term Domain Replacement Operator (FQTDR):** Replaces one domain in a **finite quantification term** by a compatible one (e.g.,, different integer sub-domain).
  - **Constant Term Replacement Operator** (CTR): Replaces a constant term by an existing term of the same type (e.g.,, Integer, Real, Complex, Char, Natural, String, Boolean).
  - **Constant Term Modification Operator** (CTM): Modifies a constant term by a user input having the same type. Although the user should provide the input, the mutant is still produced automatically.
- **Case Term Replacement Operator** (CTRO): Replaces the selected term to be executed as part of a case selection by another existing term.

Table 17: Asmetal term operators examples

Operator	Original AsmetaL code	Mutant AsmetaL code
FQTP	(exist $r in Integer with $r>0)	(exist unique $r in Integer with $r>0)
TGCR	(exist $r in Integer with $r>0)	(exist $r in Integer with $r>0)
TTR	if $value=5 then 10 endif	if $value=5 then 25 endif
ETR	if $value=5 then 10 else 20 endif	if $value=5 then 10 else 25 endif
FQTDR	(forall $v in Coordenate with isvalid($v))	(forall $v in Point with isvalid($v))
CTR	value := 10	value := 20
CTM	value := 10	value := 20 (User Input)
CTRO	switch($c) case 1 : 1 case 2 : 2 endswitch	switch($c) case 1 : 3 case 2 : 2 endswitch

4.2.4 Invariant mutation operators

Invariants are used to express constraints over functions and rules. We define the following two operators (Table **18**):

- **Invariant Condition Replacement** (ICR): Replaces the invariant condition with any existing invariant condition.
- **Invariant Declaration Deletion** (IDD): Deletes the invariant declaration statement.

Table 18: ICR and IDD operators examples

Operator	Original AsmetaL code	Mutant AsmetaL code
ICR	invariant over position: position(WO)=position(GO)	invariant over position: position(WO)!=position(GO)
IDD	invariant over position: position(WO)=position(GO)	// invariant over position: // position(WO)=position(GO)

4.2.5 Initialization mutation operators

We have defined three operators (Table **19**) to mutate AsmetaL initialization section:

- **Default Initialization Replacement Operator** (DIR): Choose a different default initialization (in case of multiple initializations) using the keyword ***default***. Only a single Optional default initialization is allowed.
- **Initialization ID Permutation Operator** (IIP): Permutes the Ids of two initialization blocks (i.e., ***init*** block).

- **Initialization Statement Deletion Operator** (ISD): Deletes a single initialization statement.

Table 19: AsmetaL initialization operators examples

Operator	Original AsmetaL code	Mutant AsmetaL code
DIR	default init s0: function signal = true function seconds = 10 init s1: function signal = false function seconds = 0	init s0: function signal = true function seconds = 10 default init s1: function signal = false function seconds = 0
IIP	init s0: function signal = true function seconds = 10 init s1: function signal = false function seconds = 0	init s0: function signal = false function seconds = 0 init s1: function signal = true function seconds = 10
ISD	init s1:	init s1:

	function signal = false function seconds = 0	function signal = false //function seconds = 0

4.3 Generation of Test Cases

For the purpose of applying mutation testing, it is necessary to generate test suites that will be the nucleus of the empirical evaluation. The used test suites must be constructed based on effective coverage criteria. In addition, the fact that test suit generation is not covered by the scope of this study, we use ATGT [89] (*a test generation tool for AsmetaL specifications that supports structural, fault based, and combinatorial coverage*) in order to generate test cases from our specification under test S. We run the obtained test suite against the set of generated mutants using the AsmetaV [90] tool. An ATGT test case, written in ASM Validation Language (AVaLLA) [90], specifies the interaction steps between the system and its environment as well as performs correctness checks (*e.g.,*, *function values*) at each step. Table 20 shows an example of AVaLLA test case, while Table 21 illustrates the results of that very test case. A given test case, part of the test suite, is said to kill a mutant if the output produced by the mutant is different from the expected output produced by the original AsmetaL specification. Hence, the test case is good enough to detect the change between the original and the mutant AsmetaL specification. It should be noted that the proposed approach is applicable for manual test case generation as well.

Table 20: AVaLLA test case example (.test)

Scenario Name	scenario UR8

Load specification under test	load ../../../TicTacToeXATGT.asm

Initial Step # 1
 	
Set and check function values
 	
Note that ***set*** function is used as update rule. In addition, ***set*** function can use some of AsmetaL constructs
 	
While ***check*** function is used as assertion function
	set userSelCol := 0;
	
	set methodCalled := USER_MOVE;
	
	check numOfMoves = 0;
	
	set userSelRow := 0;
	
	check res = PLAYING;
	
	check status = TURN_USER;

Step is used to go to the next state	step

Step # 2	set methodCalled := COMPUTER_MOVE;
	
	check numOfMoves = 1;
	
	check board(0) = CROSS;
	
	set userSelRow := 2;
	
	check status = TURN_PC;

Step is used to go to the next state	step

Step # 3	check board(1) = NOUGHT;
	
	check numOfMoves = 2;
	
	check status = TURN_USER;

Table 21: AVaLLA test case results generated by AsmetaV

** Simulation ** check succeeded: numOfMoves = 0	</State 2 (controlled)> check succeeded: board(1) = NOUGHT

check succeeded: res = PLAYING check succeeded: status = TURN_USER <State 1 (controlled)> board(0)=CROSS methodCalled=USER_MOVE numOfMoves=1 result=1 status=TURN_PC step__=1 userSelCol=0 userSelRow=0 </State 1 (controlled)> check succeeded: numOfMoves = 1 check succeeded: board(0) = CROSS check succeeded: status = TURN_PC <State 2 (controlled)> board(0)=CROSS board(1)=NOUGHT methodCalled=COMPUTER_MOVE numOfMoves=2 result=1 status=TURN_USER step__=2 userSelCol=0 userSelRow=2	check succeeded: numOfMoves = 2 check succeeded: status = TURN_USER <State 3 (controlled)> board(0)=CROSS board(1)=NOUGHT methodCalled=COMPUTER_MOVE numOfMoves=2 result=1 status=TURN_USER step__=3 userSelCol=0 userSelRow=2 </State 3 (controlled)> <State 4 (controlled)> board(0)=CROSS board(1)=NOUGHT methodCalled=COMPUTER_MOVE numOfMoves=2 result=1 status=TURN_USER step__=3 userSelCol=0 userSelRow=2 </State 4 (controlled)>

ATGT translates AsmetaL specification into Spin model-checker [91] in order to use the produced counter examples to generated test cases. ATGT provides several coverage criteria to generate test cases. It includes structural coverage such as basic rule coverage, update rule coverage, and MCDC Coverage (*see section 3.1.3).* In addition, it provides the following criteria:

- Fault-based Coverage [92]: aims at generating test cases based on fault injection in guard condition including the following operators LNF, ENF, MLF, ST0/1, ASF, ORF, and ROF.

- Pair-wise Coverage [93]: aims at validating each possible pair of input values by applying constraints over the input domain.
- Three-wise Coverage [94]: aims at validating t-wise of input values by applying constraints over the input domain, where t is equal to 3.

4.4 Analysis of the proposed operators

In this section, we characterize mathematically the upper bound of the number of produced mutants for each operator.

Number of mutant (upper bound)

Table 22 presents the upper bound for each operator.

Table 22: The upper bound for the number of generatred mutants per operator.

Operators	Upper Bound
FTP	$\lvert function\ signatures \rvert * 3$
RGCR	$\lvert rule\ guard\ conditions \rvert * (\lvert unique\ guard\ conditions \rvert - 1)$
TRR	$\lvert then\ rules \rvert * (\lvert unique\ rules \rvert - 1)$
ERR	$\lvert else\ rules \rvert * (\lvert unique\ rules \rvert - 1)$
MMR	$\lvert macro\ rule\ declarations \rvert - 1$
PB2S	$\lvert block\ rules \rvert$
S2PB	$\lvert sequence\ rules \rvert$
ARO	$(\lvert block\ rules \rvert + \lvert sequence\ rules \rvert) * \lvert unique\ rules \rvert$

RRO	$\left(\sum_{Block\ rule\ \boldsymbol{i}} \lvert rules\ in\ block\ i \rvert + \sum_{Sequence\ rule\ \boldsymbol{i}} \lvert rules\ in\ Sequence\ i \rvert \right) * \lvert unique\ rules \rvert$
SBSDL	$\sum_{block\ i} \lvert rules\ in\ Block\ i \rvert + \sum_{sequence\ i} \lvert rules\ in\ sequence\ i \rvert$
SSM	$\sum_{sequence\ rule\ \boldsymbol{i}} \frac{\lvert rules\ in\ sequence\ \boldsymbol{i} \rvert * (\lvert rules\ in\ sequence\ \boldsymbol{i} \rvert - 1)}{2}$
CDoR	$\lvert choose\ rules \rvert * (\lvert unique\ rules \rvert - 1)$
CIR	$\lvert choose\ rules \rvert * (\lvert unique\ rules \rvert - 1)$
CRE	$\lvert choose\ rules \rvert$
CDR	$\sum_{choose\ rule\ \boldsymbol{i}} \lvert domain\ elements\ in\ i \rvert * (\lvert domain\ signatures \rvert - 1)$
FDoR	$\lvert forall\ rules \rvert * (\lvert unique\ rules \rvert - 1)$
FCRP	$\lvert choose\ rules \rvert + \lvert forall\ rules \rvert$
RTS	$\lvert non\ skip\ rules \rvert$
SSSC	$\sum_{case\ term\ \boldsymbol{i}} \lvert case\ in\ \boldsymbol{i} \rvert + \sum_{case\ rule\ \boldsymbol{i}} \lvert case\ in\ \boldsymbol{i} \rvert$
SCP	$\sum_{case\ term\ \boldsymbol{i}} \frac{\lvert cases\ in\ \boldsymbol{i} \rvert * (\lvert cases\ in\ \boldsymbol{i} \rvert - 1)}{2}$

	$+ \sum_{case\ rule\ \boldsymbol{i}} \frac{\lvert cases\ in\ \boldsymbol{i}\rvert * (\lvert cases\ in\ \boldsymbol{i}\rvert - 1)}{2}$
CRRO	$\lvert case\ rules\rvert * (\lvert unique\ rules\rvert - 1)$
DSC	$\sum_{case\ rule\ \boldsymbol{i}} \lvert cases\ in\ i\rvert + \sum_{case\ term\ \boldsymbol{i}} \lvert cases\ in\ i\rvert$
LRVA	$\sum_{let\ rule\ \boldsymbol{i}} \lvert terms\ in\ i\rvert * (\lvert unique\ terms\rvert - 1)$
LRR	$\lvert let\ rules\rvert * (\lvert unique\ rules\rvert - 1)$
LRVR	$\sum_{let\ rule\ \boldsymbol{i}} \lvert variables\ in\ i\rvert * (\lvert unique\ variables\rvert - 1)$
EDR	$\sum_{extend\ rule\ \boldsymbol{i}} \lvert domains\ in\ i\rvert * (\lvert domain\ signatures\rvert - 1)$
ERRO	$\lvert extend\ rules\rvert * (\lvert unique\ rules\rvert - 1)$
EIR	$\sum_{extend\ rule\ \boldsymbol{i}} \lvert variables\ in\ i\rvert * (\lvert unique\ variables\rvert - 1)$
AOR	$\lvert arithmetic\ operators\rvert * 3$
UOI	$\lvert basic\ domains\rvert + \lvert functions\ return\ basic\ domains\rvert$
LOR	$\lvert logical\ operators\rvert * 4$
ROR	$\lvert relational\ operators\rvert * 5$

ENF	$\lvert guard\ conditions \rvert$
LNF	$\lvert boolean\ terms \rvert + \lvert function\ terms\ returning\ a\ Boolean \rvert$
STF	$\lvert guard\ conditions \rvert * 2$
ABS	$\lvert integer\ and\ double\ terms \rvert + \lvert functions\ returning\ integer\ and\ double \rvert$
FQTP	$\lvert Finite\ Quantification\ Terms \rvert * 2$
TGCR	$\lvert term\ guard\ conditions \rvert * (\lvert unique\ guard\ conditions \rvert - 1)$
TTR	$\lvert then\ terms \rvert * (\lvert unique\ terms \rvert - 1)$
ETR	$\lvert else\ terms \rvert * (\lvert unique\ terms \rvert - 1)$
FQTDR	$\sum_{finite\ quntification\ tem\ \boldsymbol{i}} \lvert domains\ in\ i \rvert * (\lvert domain\ signatures \rvert - 1)$
CTR	$\lvert boolean\ terms \rvert + \sum_{\substack{type\ \boldsymbol{i}=\{\boldsymbol{char},\boldsymbol{complex},\boldsymbol{integer},\\ \boldsymbol{natural},\ \boldsymbol{real},\boldsymbol{string},\}}} \lvert type\ i\ terms \rvert * (\lvert type\ i\ terms \rvert - 1)$
CTM	$\sum_{\substack{type\ \boldsymbol{i}=\{\boldsymbol{char},\boldsymbol{complex},\boldsymbol{integer},\\ \boldsymbol{natural},\ \boldsymbol{real},\boldsymbol{string},\}}} \lvert type\ i\ terms \rvert$
CTRO	$\lvert case\ terms \rvert * (\lvert unique\ terms \rvert - 1)$

ICR	$\lvert invariants \rvert \, * \, (\lvert unique\ guard\ conditions \rvert \, - \, 1)$
IDD	$\lvert invariants \rvert$
DIR	$\lvert initializations \rvert - 1$
IIP	$\frac{(\lvert initializations \rvert * (\lvert initilizations \rvert - 1))}{2}$
ISD	$\lvert initialization\ statments \rvert$

4.5 Chapter Summary

In this chapter, the proposed approach methodology was presented briefly including design of AsmetaL mutation operators, AsmetaL mutation tool, empirical evaluation, and selective mutation testing. In addition, the set of proposed AsmetaL mutation operators was reviewed in full details. Moreover, the test case generation criteria provided by ATGT was presented.

CHAPTER 5

MuAsmetaL: An AsmetaL Mutation Experimental Tool

MuAsmetaL (M*utation testing system for AsmetaL*) is an integrated framework that facilitates the generation and validation of mutants, and the execution of test cases against mutants for AsmetaL specifications. It integrates several AsmetaL tools (*AsmetaLc, AsmetaV, and AsmetaS*) used to preform automatic mutation testing. MuAsmetaL is a prototype tool developed as a proof of concept of our proposed mutation testing/mutation operators for AsmetaL language. We intend to public release the final version [106] to help practitioners and researchers.

5.1 Tool Requirements

In order to apply mutation testing on AsmetaL specifications, we have elicited the following minimal requirements for MuAsmetaL support:

R1 Creating and saving of new AsmetaL specifications (.asm files).

R2 Opening and editing of existing AsmetaL specification.

R3 Visualizing AsmetaL specifications using syntax highlights.

R4 Generating mutants based on user selection of a set of operators to be applied.

R5 Validating the correctness of the generated mutants using AsmetaLc.

R6 Validating syntactic equivalency of generated mutants against the original specification.

R7 Viewing mutants.

R8 Importing and viewing test cases.

R9 Running test cases against the original specification using AsmetaV.

R10 Running test cases against the generated mutants using AsmetaV.

R11 Generating test report (.csv files) contains a table that shows the status mutants against test cases e.g., pass, or fail.

R12 Simulating the original specification using AsmetaS.

R13 Simulating the generated mutants using AsmetaS.

R14 Calculating mutation score per operator and for all mutants.

The MuAsmetaL tool fulfills the aforementioned requirements while providing a user-friendly interface.

5.2 MuAsmetaL Architecture

MuAsmetaL is implemented using Java. MuAsmetaL incarnates the following:

- **AsmetaLc [24]** is used to syntactically validate the specifications (*original/mutants*).
- **AsmetaV [26]** runs specifications (*original/mutants*) against test cases (*AVaLLA*).
- **AsmetaS [25]** simulates the execution of specifications (*original/mutants*)

Figure 11 shows the general architecture of MuAsmetaL tool. It is decomposed into five main components (*editor, parser, data structure, mutation engine, and tester*).

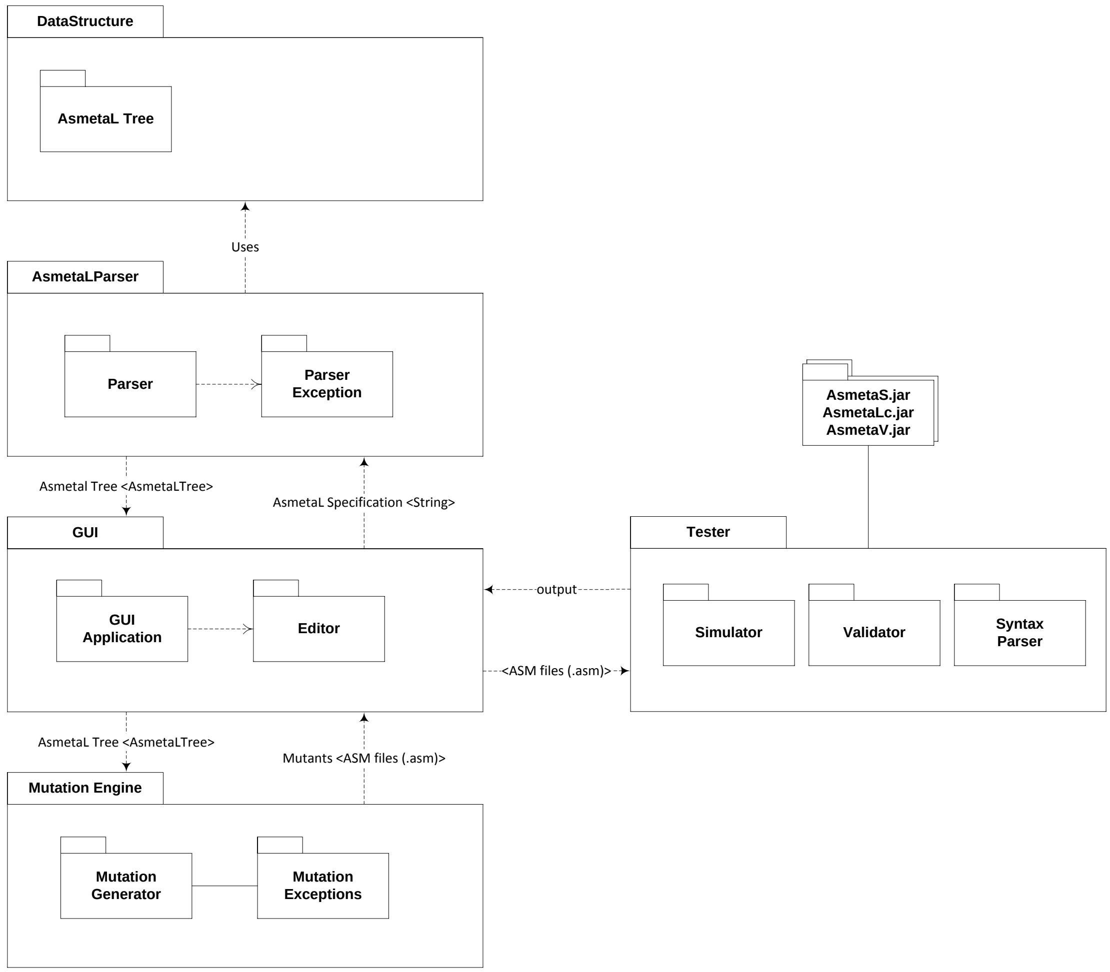


Figure 11: MuAsmetaL Structure

Editor

The Editor component provides a graphical user interface for MuAsmetaL that handles opening and saving of AsmetaL (.asm) files. In addition, it provides a syntax highlight and a simple autocomplete mechanism. It relies on JTextPane component with custom document style to view and highlight the AsmetaL syntax. Moreover, visualizer component is implemented, it takes String as input and illustrates tree using JPanel. The editor component provides a simple auto complete mechanism based on AsmetaL keywords. This component fulfils the requirements **R1**, **R2**, **R3**, **R7**, and **R8**.

Parser

The parser supports all AsmetaL constructs defined by the EBNF grammar. It is generated using javacc tool [107][107]. The input for the parser is either an AsmetaL specification file (.asm) or an AsmetaL specification described as a String, while the output is an ASMetaLTree.

Tree Data Structure

MuAsmetaL implements a comprehensive data structure that follows the AsmetaL Language grammar [*EBNF*]. It is described as a tree *called AsmetaLTree* (see Figure 12). **AsmetaLTree** has 132 different node types. The root and its children follows the main structure of AsmetaL (see Figure 3), while the rest of the tree is dynamic structure based on the specification structure. ASMetaLTree provides a manifest object (*contains sets of pointers for each node type in order to facilitate the traversing of tree with dynamic structure e.g.,set of rules, set of terms*).

Moreover, ASMetaLTree can be deeply cloned. Indeed, a new tree version is generated for each mutant. In addition, the AsmetaLTree supports comparable interfaces, in which any two nodes can be compared with each other and their children recursively. This comparison feature allows syntactic equivalency between the original specification and the mutant.

Moreover, the resulting AsmetaLTree is used to generate AsmetaL syntax for mutants (.asm files). **R6** is fulfilled by tree structure.

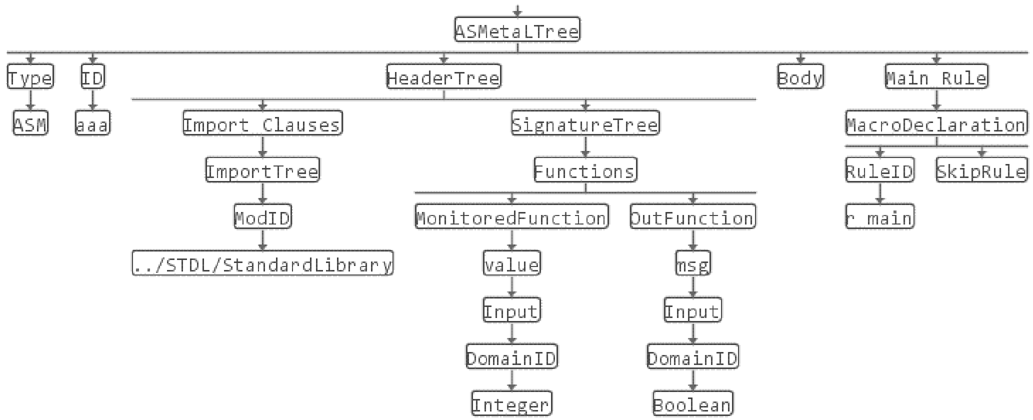


Figure 12: Example of an AsmetaL Tree

Mutation Engine

The mutation engine is responsible of injecting faults into AsmetaL specifications by applying all mutation operators. The input of the mutation engine is an ASMetaLTree, while the output is one or many AsmetaL specification files (.asm) corresponding to the generated mutants. In addition, the mutation engine is responsible for performing syntax validation and syntactic equivalency checks as part of the mutant generation process. First, a new ASMetaLTree is generated by cloning the original tree. Then, a mutation operator is applied to the cloned tree. Next, the conformance of the mutated tree is checked against the language grammar is performed using AsmetaLc. Although, mutation operators are supposed to produce mutants that are syntactically different from the original specifications, a syntactic equivalency check is performed to make sure that the produced mutants are unique. Any mutated tree that fails the validation process is discarded. The rest of mutants will be stored as AsmetaL specification files (.asm). Requirement **R4** is fulfilled by the mutation engine component.

Tester

The tester component is responsible of validating the correctness of AsmetaL specifications (original/mutants) using AsmetaLc. The input to the AsmetaL specifications validation is (.asm file), while the output is true or false with message that indicates the location of invalid segment of specification and the expected segment of specification. In addition, it perform the execution of AsmetaL specifications (original/mutants) using AsmetaS. The input to AsmetaL specification execution is (.asm file), while the output is the execution output in runtime in form of String. Moreover, the actual test (running test cases against specifications) is done by the tester component in which it relies on AsmetaV. The input for the AsmetaL testing is (.asm file) and the test suite, while the output is either Pass, Fail, or Runtime Exception.

The Tester fulfils the requirements **R5**, **R9**, **R10**, **R11**, **R12**, **R13**, and **R14**.

5.3 MuAsmetaL in Practice

In this section, we describe the purpose of our tool and how it can be used to generate and execute AsmetaL mutants. Let us consider the following example (see Figure 13, integer absolute value specification) to show the usefulness of our proposed tool.

```
asm absolutevalue
import ../STDL/StandardLibrary
signature:
monitored value:Integer
controlled output:Integer
definitions:
main rule r_main =
      if(value<0) then
            output := value*-1
      else
            output := value
      endif
```

Figure 13: Absolute value AsmetaL specification

In this section, we provide several screenshots that show how the aforementioned requirements are fulfilled:

R1 Creating AsmetaL specification file (.asm).

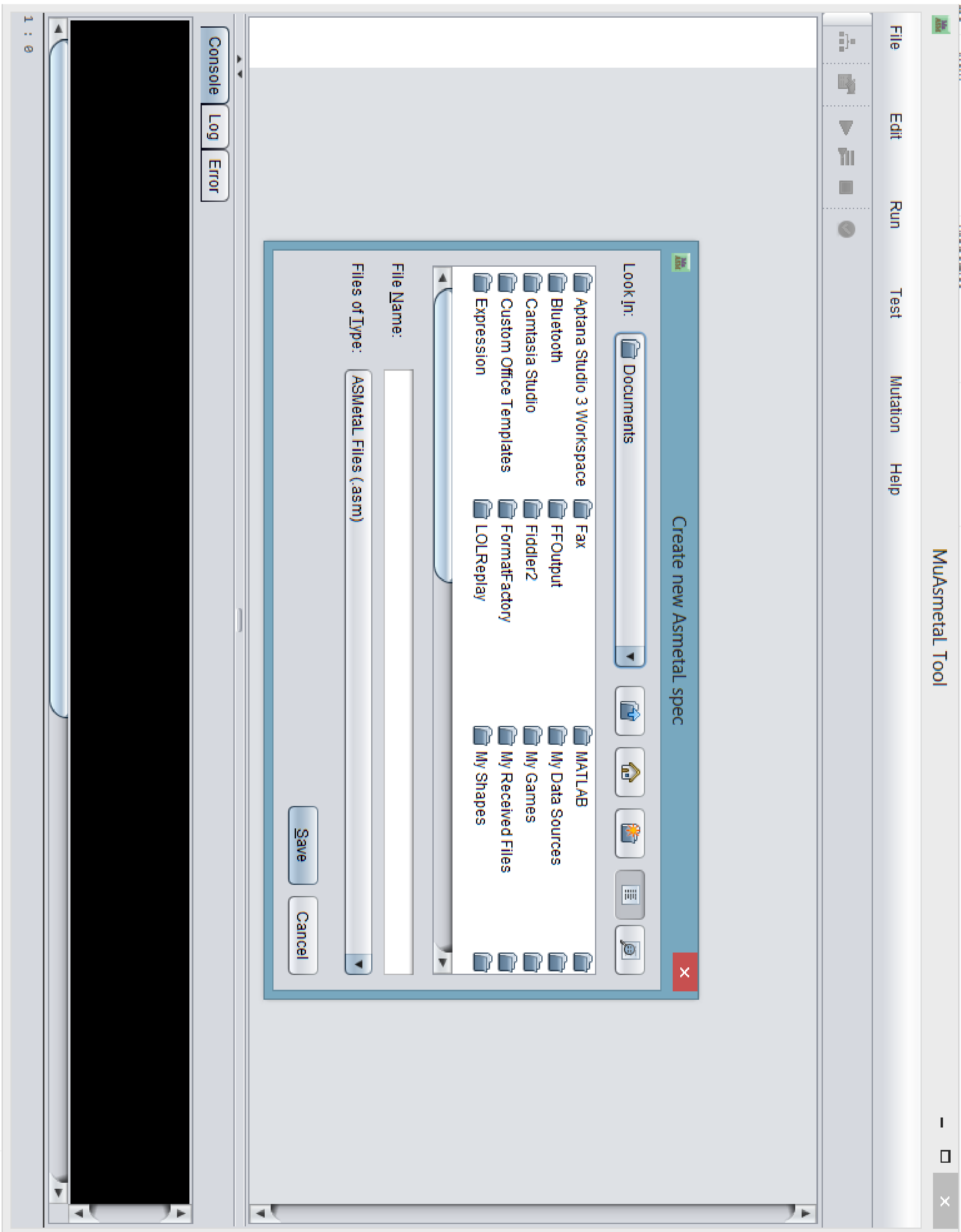


Figure 14: Creating new AsmetaL specification using MuAsmetaL

R2 Editing new/existing AsmetaL specification.

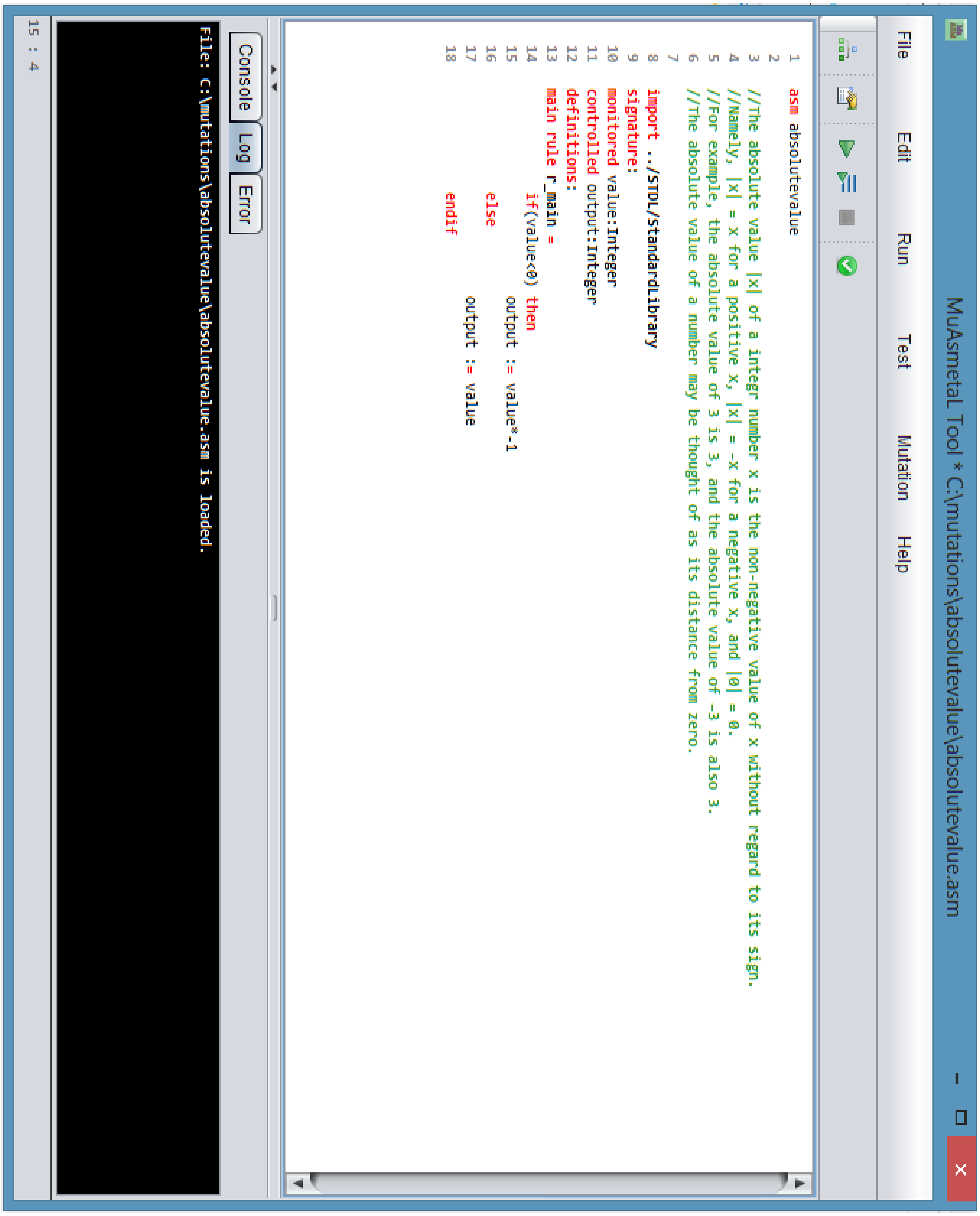


Figure 15: Editing existing AsmetaL specification using MuAsmetaL

R3 Visualizing original specification.

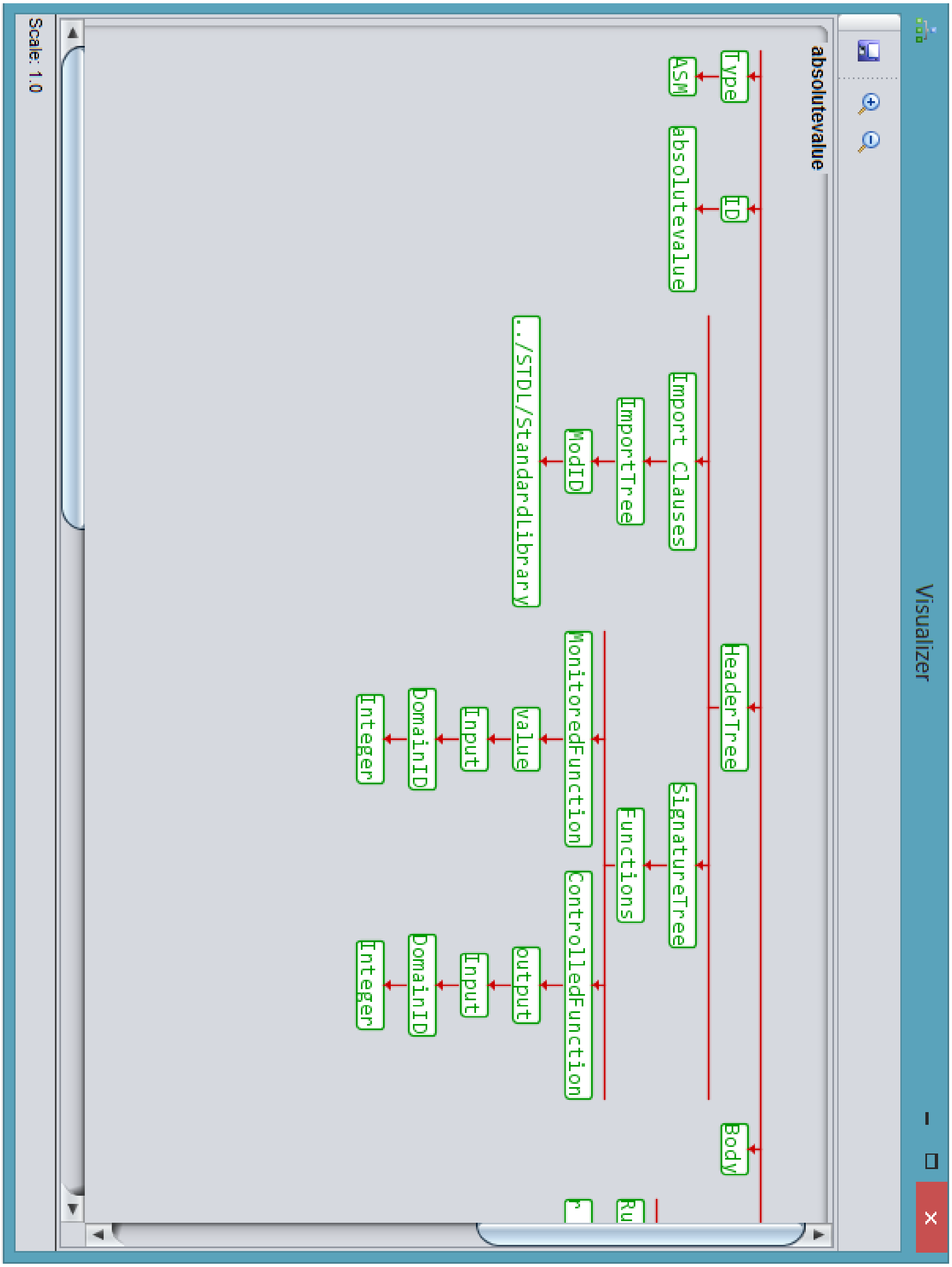


Figure 16: Visualizing ASMetaLTree using MuAsmetaL

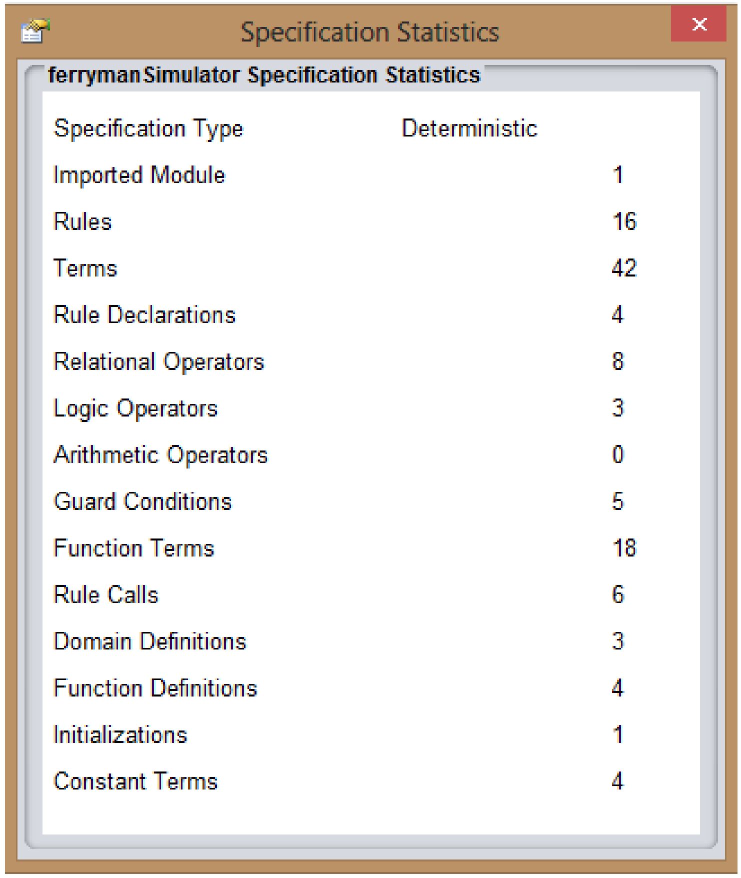


Figure 17: Statistical information about AsmetaL Specification using MuAsmetaL

R4 Generating mutants based on the proposed operators.

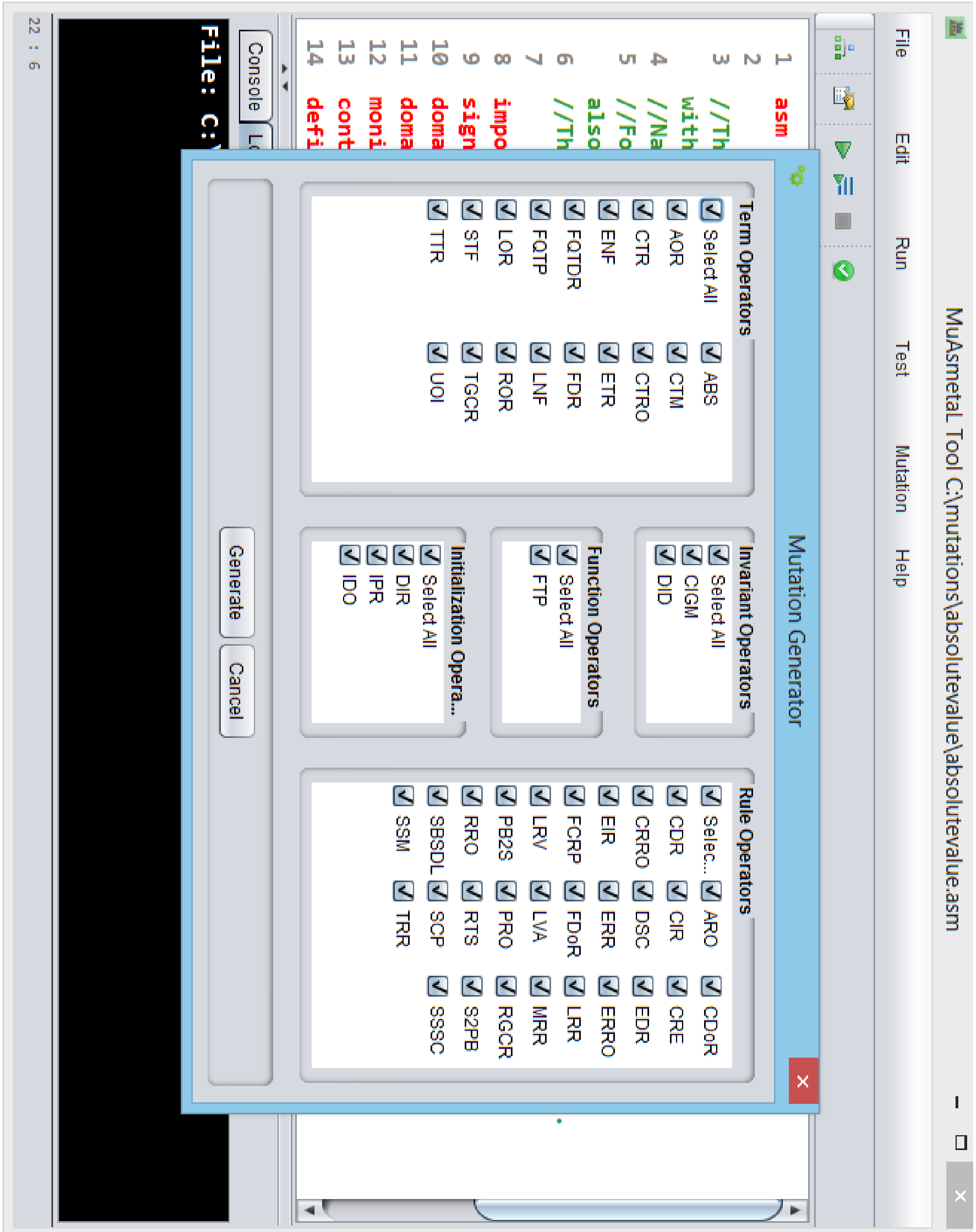
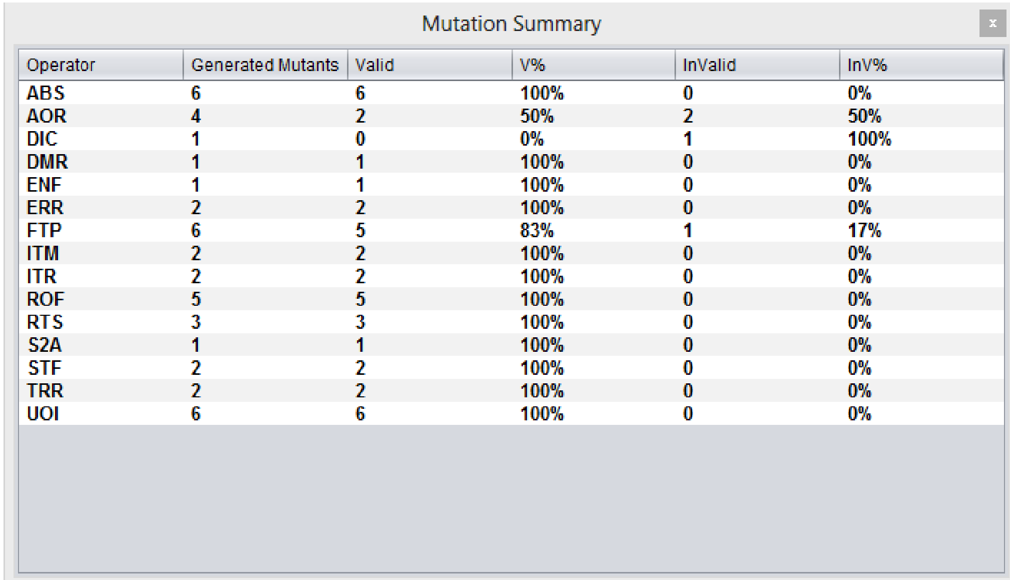


Figure 18: MuAsmetaL mutation generation interface



Mutation Summary

Operator	Generated Mutants	Valid	V%	InValid	InV%
ABS	6	6	100%	0	0%
AOR	4	2	50%	2	50%
DIC	1	0	0%	1	100%
DMR	1	1	100%	0	0%
ENF	1	1	100%	0	0%
ERR	2	2	100%	0	0%
FTP	6	5	83%	1	17%
ITM	2	2	100%	0	0%
ITR	2	2	100%	0	0%
ROF	5	5	100%	0	0%
RTS	3	3	100%	0	0%
S2A	1	1	100%	0	0%
STF	2	2	100%	0	0%
TRR	2	2	100%	0	0%
UOI	6	6	100%	0	0%

Figure 19: MuAsmetaL mutation generation summary

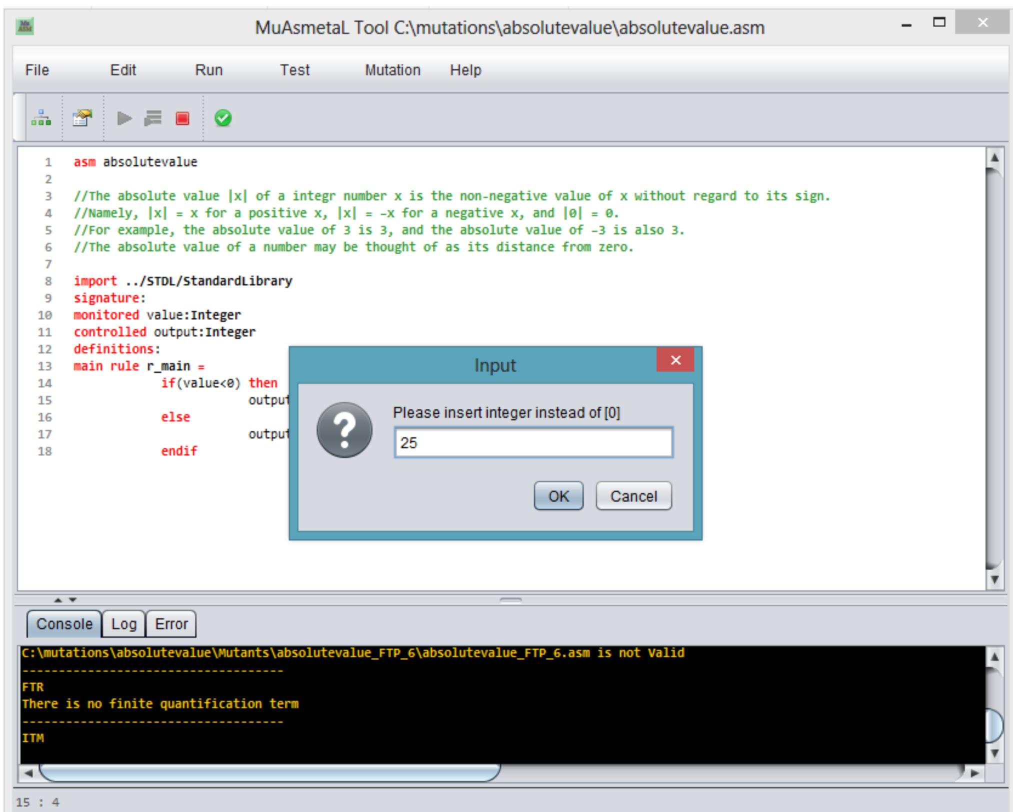


Figure 20: MuAsmetaL handles manual input from the user

R5 Validating the correctness of all the generated mutants using AsmetaLc.

R6 Validating syntactic equivalency of generated mutants against the original specification.

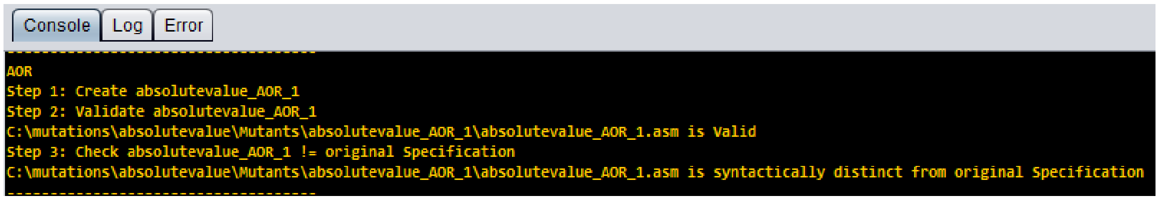


Figure 21: AsmetaL specification correctness validation and syntactic equivalency validation

R7 Viewing mutants.

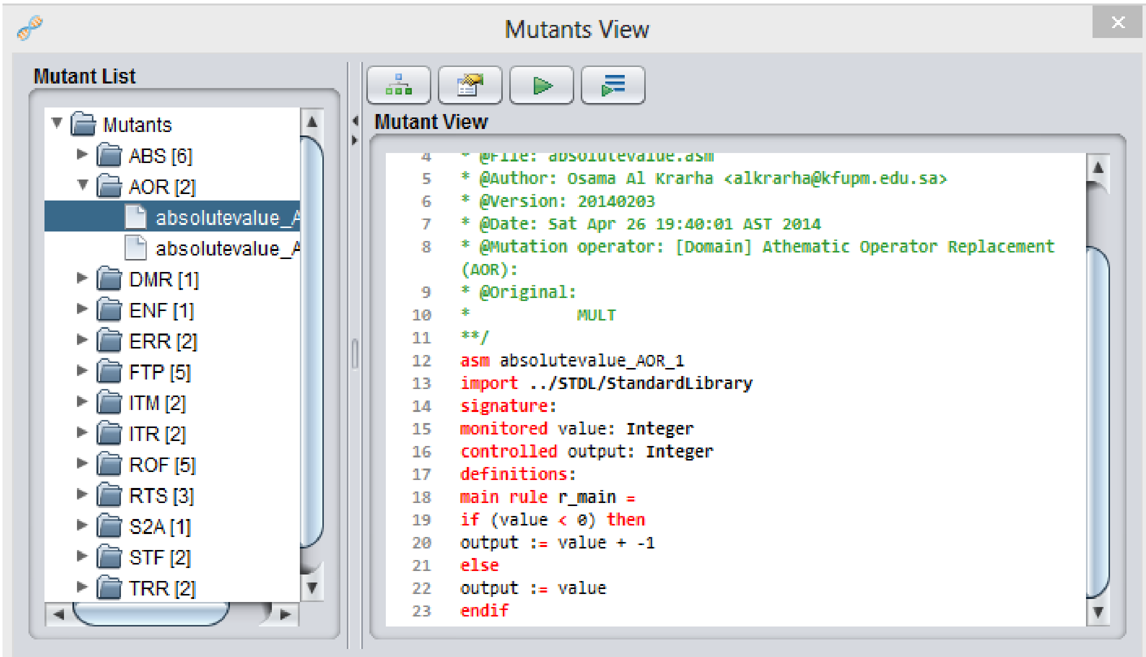


Figure 22: MuAsmetaL mutants' viewer

R8 Importing test cases.

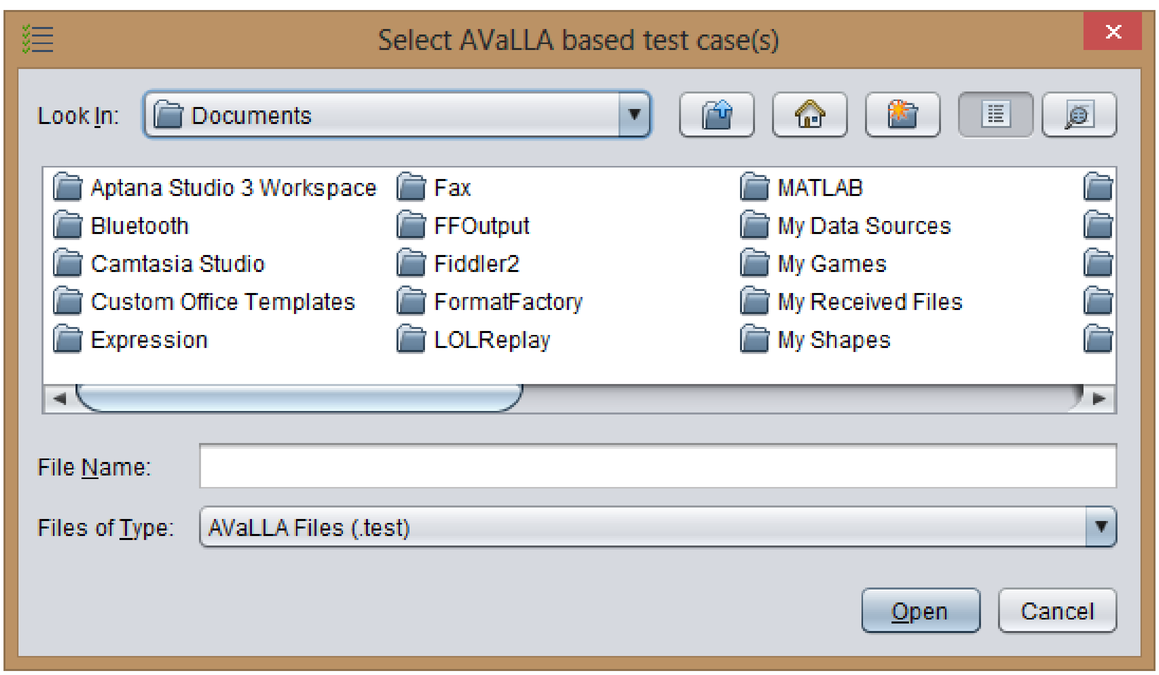


Figure 23: Import AVaLLA test cases using MuAsmetaL

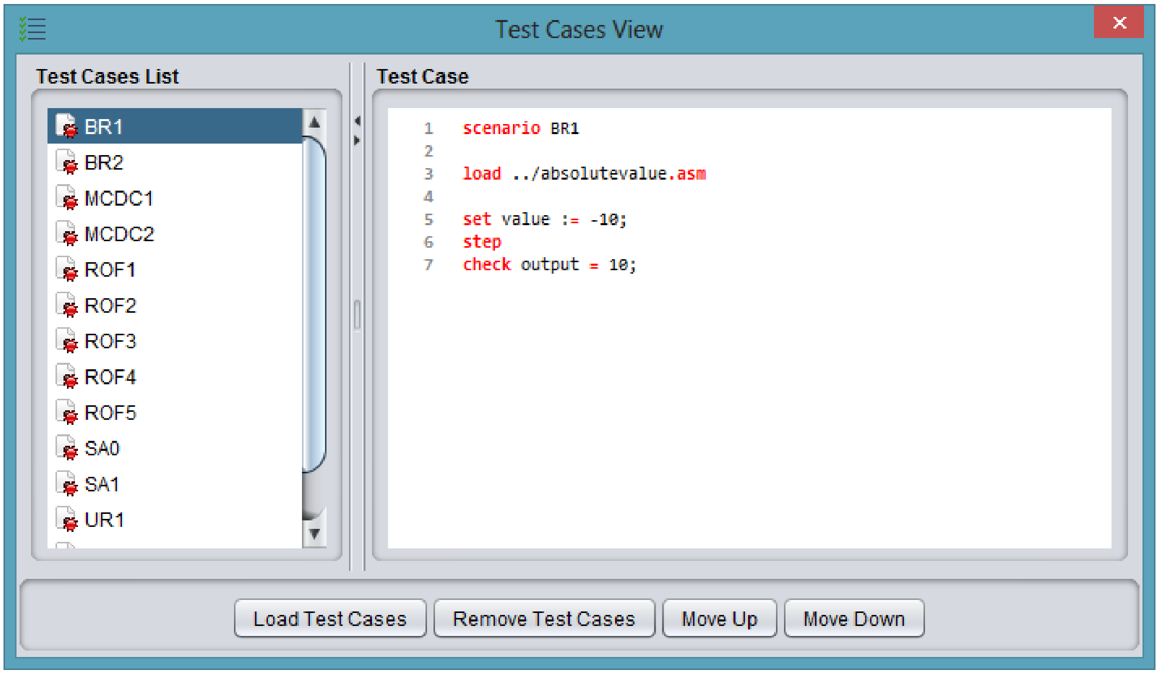


Figure 24: Viewing/Ordering test cases using MuAsmetaL

R9 Running test cases against the original specification using AsmetaV.

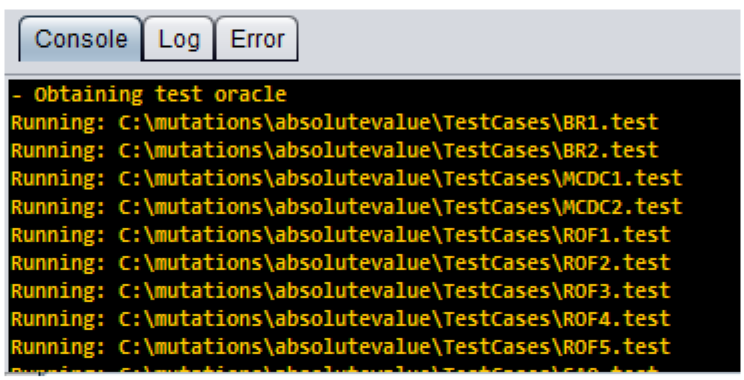


Figure 25: Running test cases against original Specification using MuAsmetaL to obtain test oracles

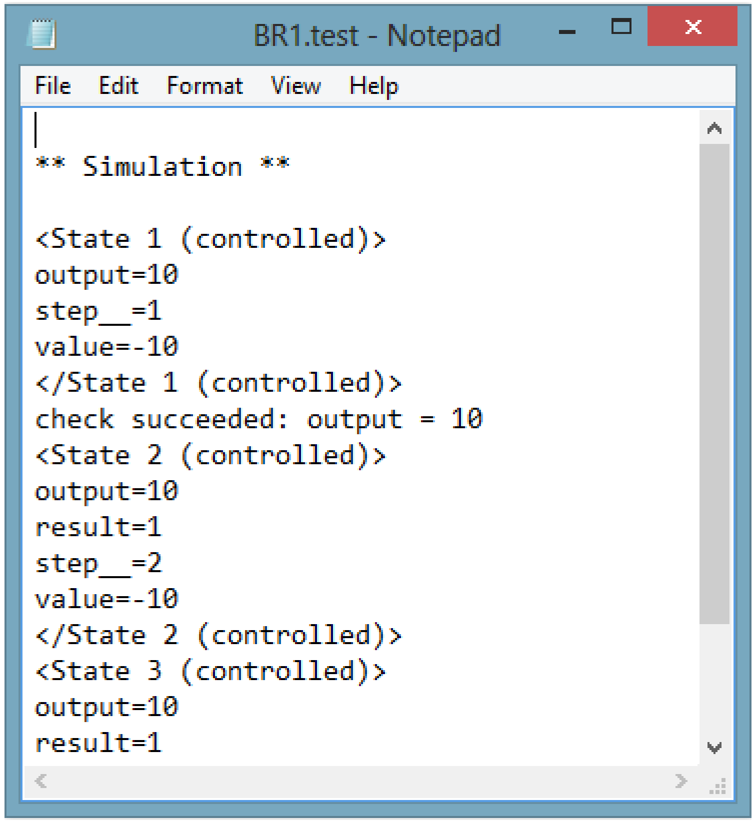


Figure 26: Test case results

R10 Running test cases against mutants using AsmetaV.

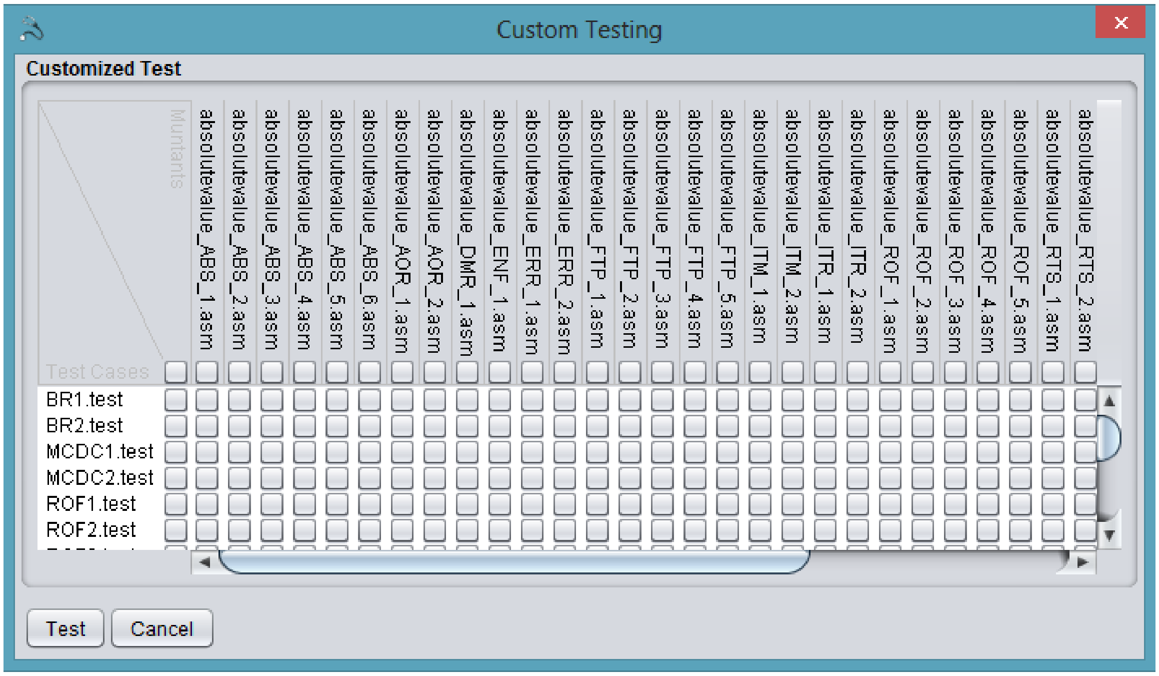


Figure 27: MuAsmetaL custom testing

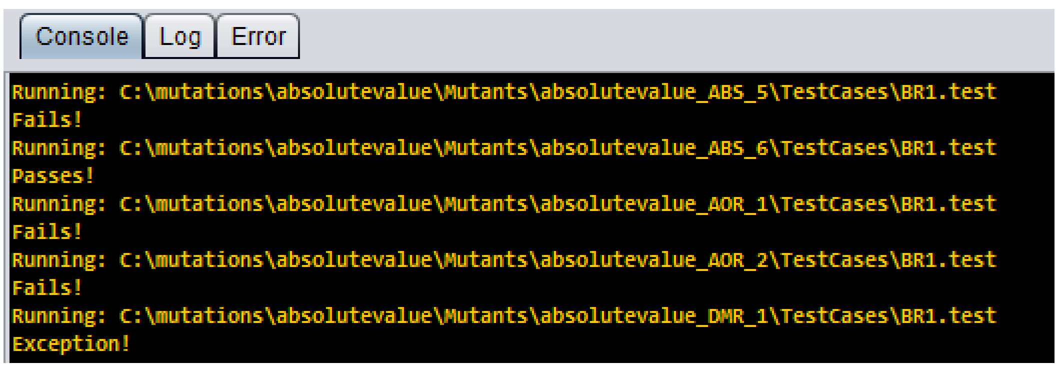


Figure 28: Running test cases against mutants

R11 Generating test report files (.csv).

	absolutev	absolutev	absolutev	absolutev	absolutev	absolutev	absolutev	absolutev	absolutev
BR1.test	F	P	F	P	F	P	F	F	E
BR2.test		P		P		P			
MCDC1.test		P		P		P			
MCDC2.test		P		P		P			
ROF1.test		P		P		P			
ROF2.test		P		P		P			
ROF3.test		P		P		P			
ROF4.test		P		P		P			
ROF5.test		P		P		P			
SA0.test		P		P		P			
SA1.test		P		P		P			
UR1.test		P		P		P			
UR2.test		P		P		P			
VNF.test		P		P		P			

Figure 29: Report file (CSV) generated by MuAsmetaL

R12 Simulating the original specification using AsmetaS.

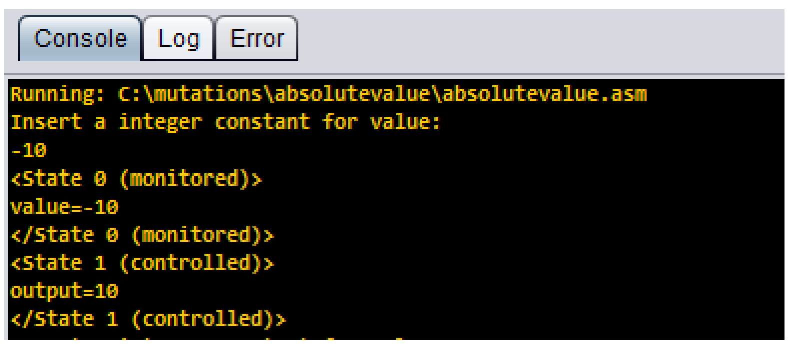
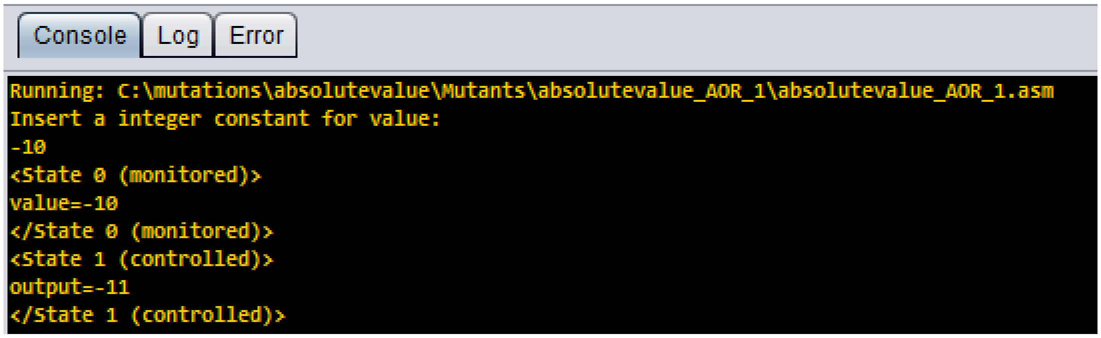


Figure 30: Simulating AsmetaL specification using MuAsmetaL

R13 Simulating mutants using AsmetaS.



R14 Calculate mutation score per operator and for all mutants.

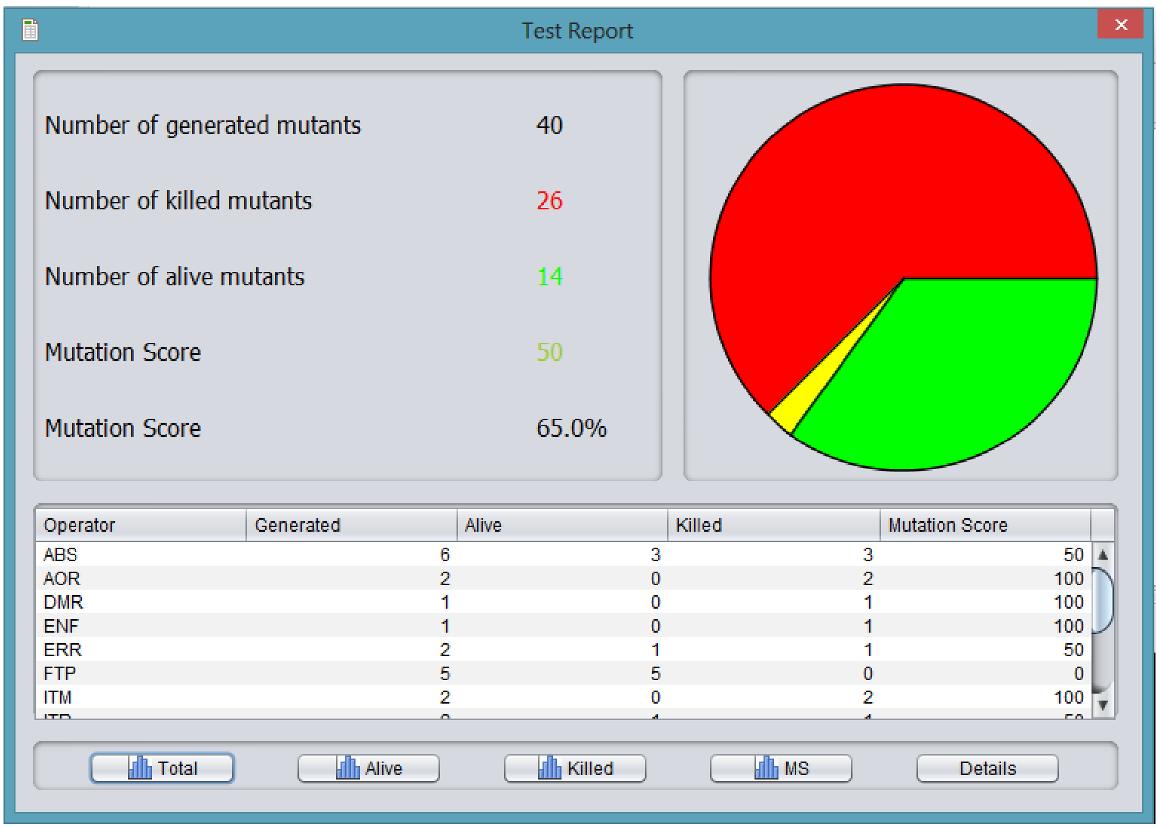


Figure 31: MuAsmetaL mutation testing results 1

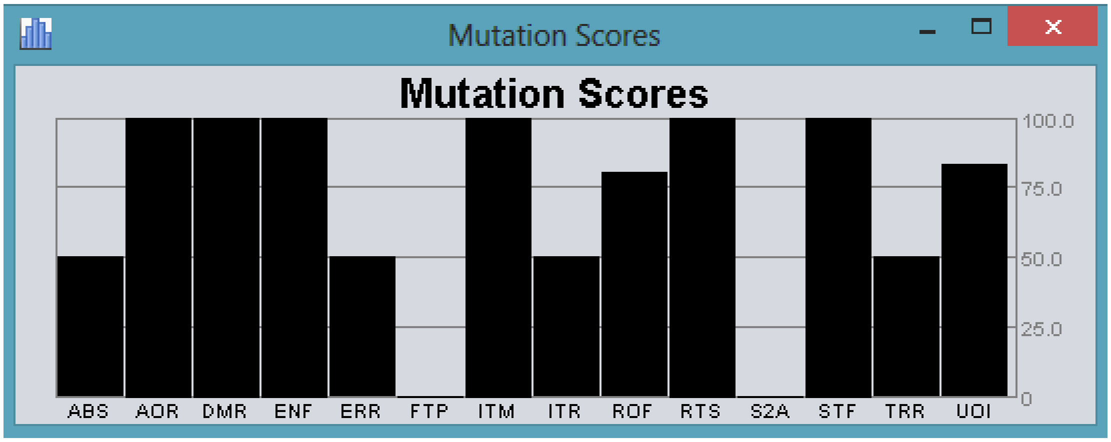


Figure 32: MuAsmetaL mutation testing results 2

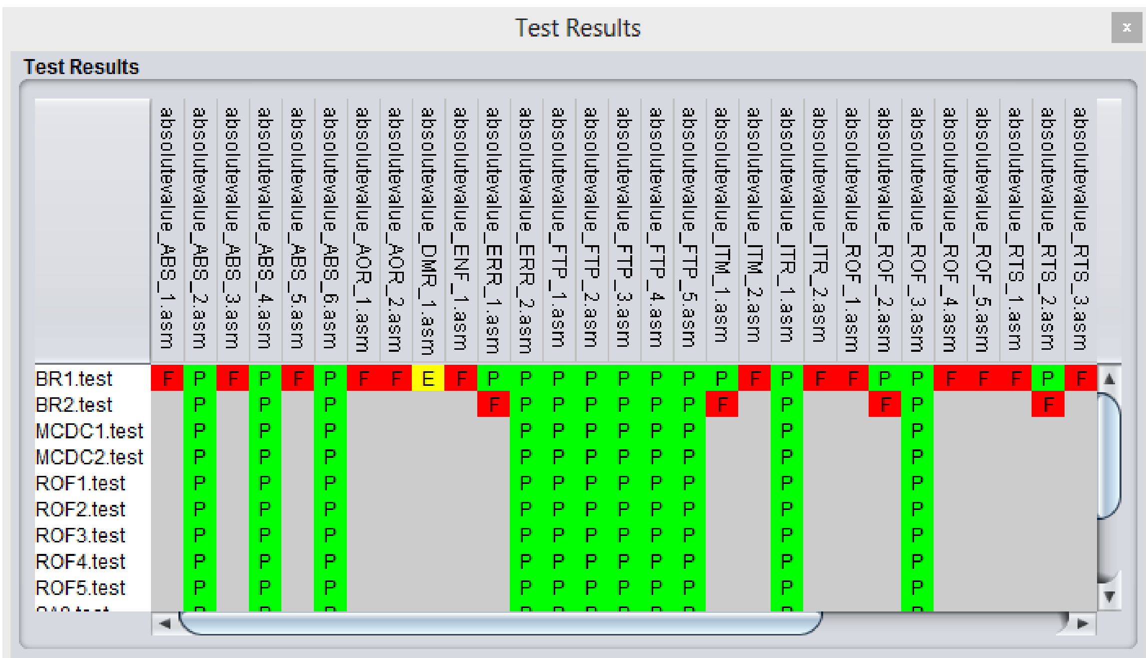


Figure 33: MuAsmetaL mutation testing results 3

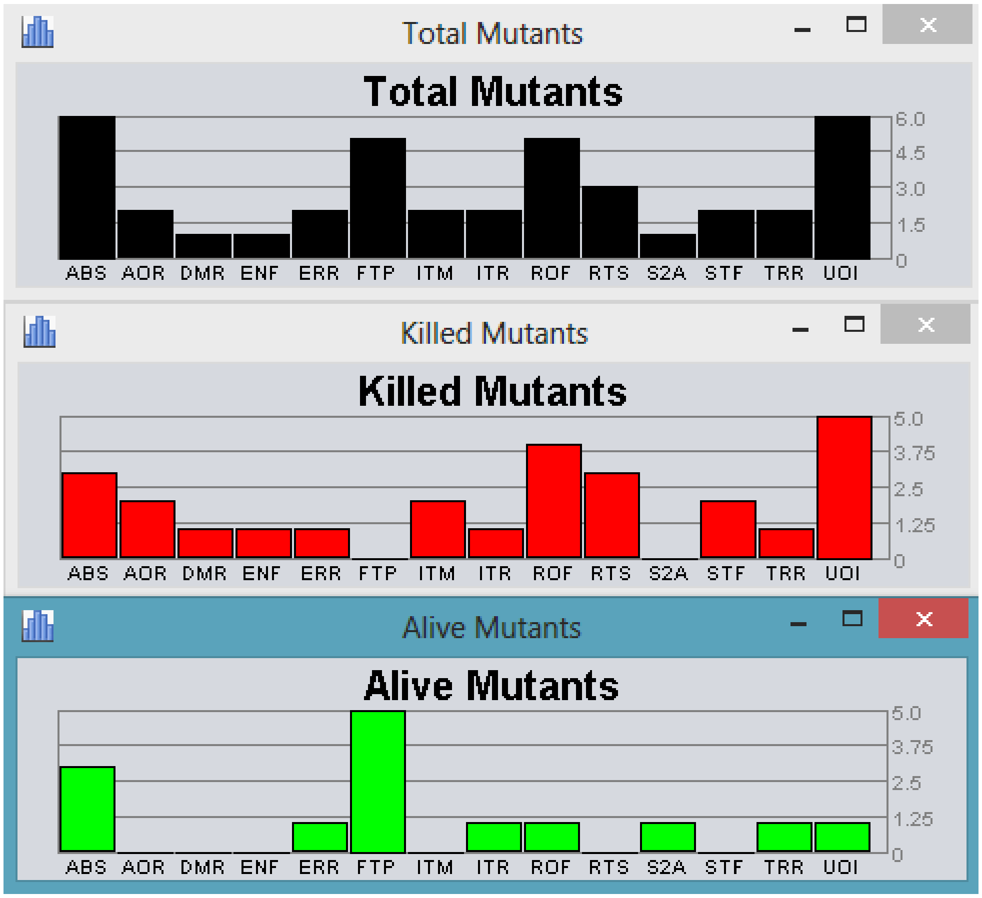


Figure 34: MuAsmetaL mutation testing results 4

5.4 Benchmarking the MuAsmetaL tool

In order to measure the performance and the mutation capabilities of our tool, we have conducted some experiments over the case studies introduced in section 6.1.

Table 23 summarizes the time spent to generate and validate mutants.

Table 23: Time spent to generate and validate mutants per case study

	Number of Mutants	Duration in Seconds	Average Time per Mutant
ferrymanSimulator	280	613 s	2.19s/m

railroadGate	193	349 s	1.81s/m
sluiceGateGround	203	281 s	1.38s/m
cruiseControl	421	552 s	1.31s/m
AdvancedClock	210	448 s	2.13s/m
AdvancedClock2	204	387 s	1.9s/m
fattoriale	136	231 s	1.7s/m

Table 24 summarizes the time spent to execute test cases and generate reports.

Table 24: Time spent to execute test cases and generate reports per case study.

	Number of Test Cases	Number of Mutants	Number of Intersection	Ratio of Intersection	Duration in Seconds	Average Time per Intersection
ferrymanSimulator	64	280	4025	22.46%	5520 s	1.37s/i
railroadGate	77	193	6174	41.54%	13260 s	2.15s/i
sluiceGateGround	46	203	4958	53.09%	12540 s	2.53s/i
cruiseControl	102	421	18858	43.92%	29400 s	1.56s/i
AdvancedClock	1	210	210	100%	420 s	2s/i
AdvancedClock2	45	204	2896	31.55%	4680 s	1.62s/i

fattoriale	44	136	3135	52.39%	5100 s	1.63s/i

5.5 MuAsmetaL Limitation

MuAsmetaL presents the following limitations:

1. **MuAsmetaL does not generate test cases.**

   However, it supports importing *AVaLLA* test cases generated by ATGT or manually.

2. **MuAsmetaL does not preform equivalency analysis.**

   Since the scope of the proposed approach does not include semantic equivalency analysis, MuAsmetaL does not perform any semantic equivalency analysis and it depends on the analyst to perform it manually. In addition, the mutation score is calculated without any consideration of equivalent mutants.

   It is worth noting that MuAsmetaL is still in prototype stages and requires testing and documentation.

5.6 Chapter Summary

This chapter shows all the details of the design and development of MuAsmetaL tool. These details include the tool requirements, and the general architectural and structure of tool, tool workflow. In addition to measuring the tool performance of tool based on benchmarks. Finally, we list limitations of the tool.

CHAPTER 6

Empirical Evaluation of the AsmetaL-based Mutation Operators

In this Chapter, we evaluate empirically the proposed suite of AsmetaL mutation operators, introduced in chapter 4, by applying them to seven different case studies. In addition, this experiment aims at assessing both the effectiveness of the proposed operators and the adequacy of test suites produced by ATGT tool and test cases that are manually generated.

6.1 Description of the AsmetaL Case Studies

In the following sections, we present the description of 7 AsmetaL specifications that are used in our empirical study. Table 25 shows the summary of case studies structure.

Table 25: Case studies summary

	ferrymanSimulator	railroadGate	sluiceGateGround	cruiseControl	AdvancedClock	AdvancedClock2	fattoriale
Domains	3	3	2	2	3	3	0
Functions	4	6	4	6	3	4	4
Rules Deceleration	4	1	3	1	2	2	2

Invariants	2	3	0	2	0	0	0

6.1.1 Case Study 1: ferrymanSimulator Specification

ferrymanSimulator [108] specification mimics the story of a man who has a wolf, a goat, and a cabbage. The man wants to convey them across the river with his boat, which only has room for a single item only (*or without*) in a single trip. The dilemma lies in the fact that the wolf and the goat must not be on the same side of the river while the man on the other side. Moreover, the goat and the cabbage must not be on the same side while the man on the other side. Invariants are used to monitor the occurrences of these two conditions.

ferrymanSimulator specification has 3 enum domains, 4 functions (2 controlled, a monitored, and a derived), and 4 macro rules. In addition, it has 2 invariants over position function. The application of MuAsmetaL tool based on all of the proposed mutation operators resulted in 280 valid mutants. A set of 64 test cases that covers all possible input sequence combination for four steps (input: Wolf, Goat, Cabbage, and None).

Table 26: ferrymanSimulator specification mutation results

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	19	12	7	0	37%	**PB2S**	2	0	0	2	*
ICR	4	0	4	0	100%	**ROR**	8	0	8	0	100%
RGCR	5	0	5	0	100%	**RRO**	42	14	28	0	67%
CRRO	36	4	32	0	89%	**RTS**	16	0	16	0	100%
TGCR	3	1	2	0	67%	**SBSDL**	3	0	3	0	100%
IDD	2	0	2	0	100%	**SCP**	6	0	6	0	100%
DSC	4	0	4	0	100%	**SSSC**	4	0	4	0	100%
ENF	5	0	5	0	100%	**STF**	10	0	10	0	100%
ERR	3	1	2	0	67%	**CTM**	4	0	4	0	100%
ETR	54	4	50	0	93%	**CTR**	4	0	4	0	100%
FTP	7	0	0	7	*	**TRR**	11	2	9	0	82%

LOR	12	2	10	0	83%	**TTR**	13	2	11	0	85%
MMR	3	0	3	0	100%	**Total**	280	42	229	9	85%

Table 26 shows the results of applying mutation testing for ferrymanSimulator specifications. The acquired MS is 85%. Figure 35 is a visual representation of the results.

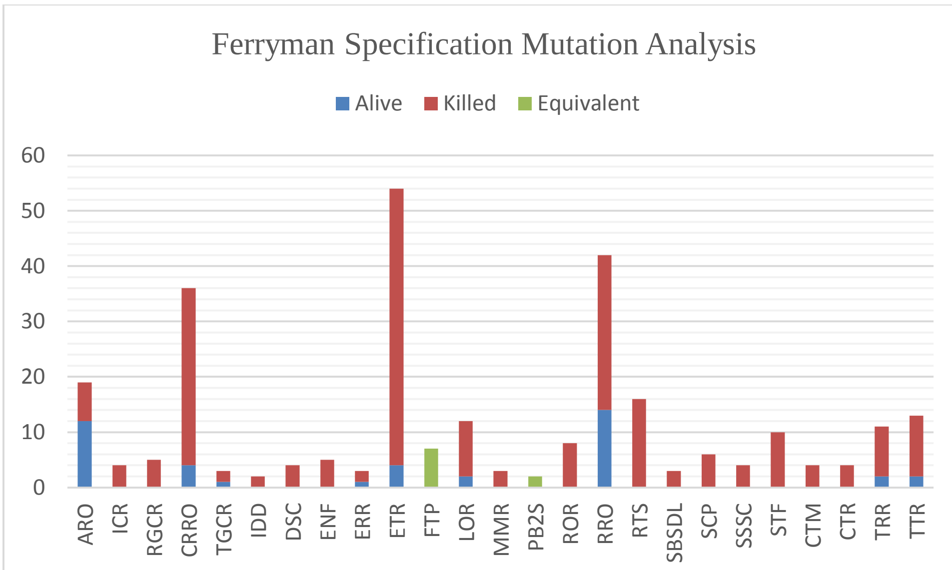


Figure 35: ferrymanSimulator specification mutation testing results

6.1.2 Case Study 2: railroadGate Specification

railroadGate [109], [110] specification describes a railroad gate system that consists of a gate and a light. The light state can be either in flashing or off state. The gate maybe closed, opened, closing, or opening states. The operation cycle starts with gate state being open and the light being off. Before the gate closes, during the closing, and until the gate is open, the light must continuously flash to warn the motorists of the closing gate. The user input is used to simulate the controlling signal that controls the light and the gate.

railroadGate specification has 3 enum domains, 6 functions (3 controlled, and 3 monitored), a macro rule. Moreover, it has 3 invariants over gate and light. MuAsmetaL tool produces 193 valid mutants. 77 test cases were generated using ATGT tool.

Table 27: railroadGate specification mutation testing results

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	7	4	3	0	43%	**LNF**	4	0	4	0	100%
ICR	12	2	10	0	83%	**PB2S**	1	0	1	0	100%
RGCR	8	0	8	0	100%	**ROR**	19	3	16	0	84%
IDD	3	1	2	0	67%	**RRO**	12	4	8	0	67%
ENF	5	0	5	0	100%	**RTS**	7	1	6	0	86%
ERR	12	5	7	0	58%	**STF**	10	1	9	0	90%
FTP	18	0	0	18	*	**TRR**	12	4	8	0	67%
LOR	60	11	49	0	82%	**CTR**	3	0	3	0	100%
						Total	193	36	139	18	79%

As shown in Table 27, and Figure 36, the resulting mutation score for railroadGate specification is 79%.

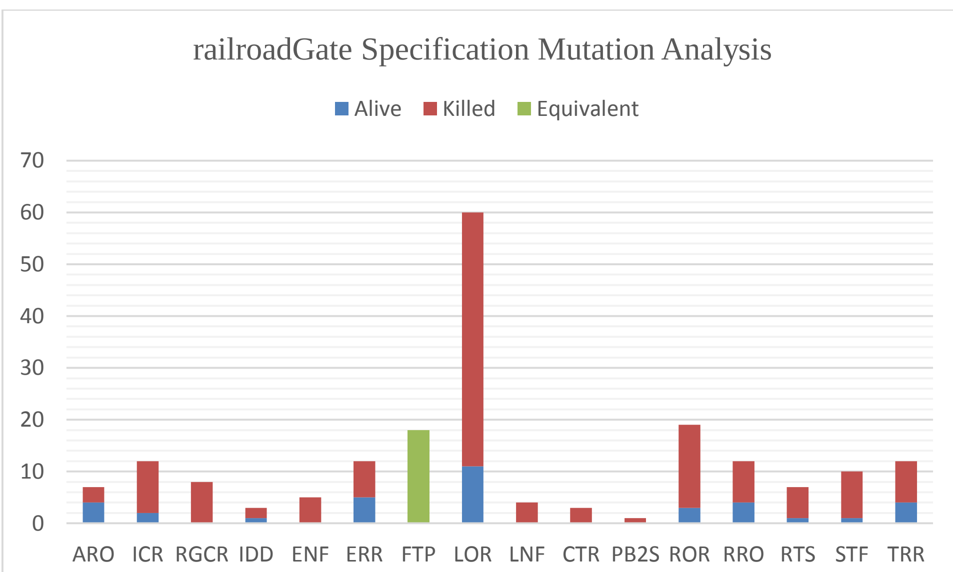


Figure 36: railroadGate specification mutation testing results

6.1.3 Case Study 3: sluiceGateGround Specification

sluiceGateGround [111], [112] specification is a ground model for simulating an irrigation system which consists of a sluice gate and a motor that opens and closes by rotating clockwise and anti-clockwise. The state of the motor can be on or off while the rotation direction can be clockwise or anti-clockwise. The motor is linked to two sensors that indicate fully opened and fully closed. The operating cycle begins by a closed sluice gate, after 170 minutes (closing period) have passed. Sluice gate starts to open until it reaches a fully opened state then wait for 10 minutes (Opening period) to pass. Then starts closing until it reaches a full closed state and then the cycle begins again.

sluiceGateGround model AsmetaL has sub-domains, an enum domain, 4 functions (2 static, a controlled, and a monitored), and 3 macro rules declarations. Using MuAsmetaL tool, 203 valid mutant were automatically generated. The generated mutant were run against 46 test cases created using ATGT tool.

Table 28: sluiceGateGround specification mutation testing results

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ABS	2	0	0	2	*	**MMR**	2	0	2	0	100%
ARO	36	24	12	0	33%	**PB2S**	3	0	1	2	100%
RGCR	12	0	12	0	100%	**ROR**	2	0	2	0	100%
ENF	4	0	4	0	100%	**RRO**	66	26	40	0	61%
FTP	5	0	0	5	*	**RTS**	11	2	9	0	82%
CTM	2	0	2	0	100%	**STF**	8	0	8	0	100%
CTR	2	0	2	0	100%	**TRR**	44	12	32	0	73%
LNF	2	0	2	0	100%	**UOI**	2	0	2	0	100%
						Total	203	64	130	9	67%

Table 28 shows results of applying mutation testing to sluiceGateGround specification, in which MS is 67%. Figure 37 provides a visual representation of the results.

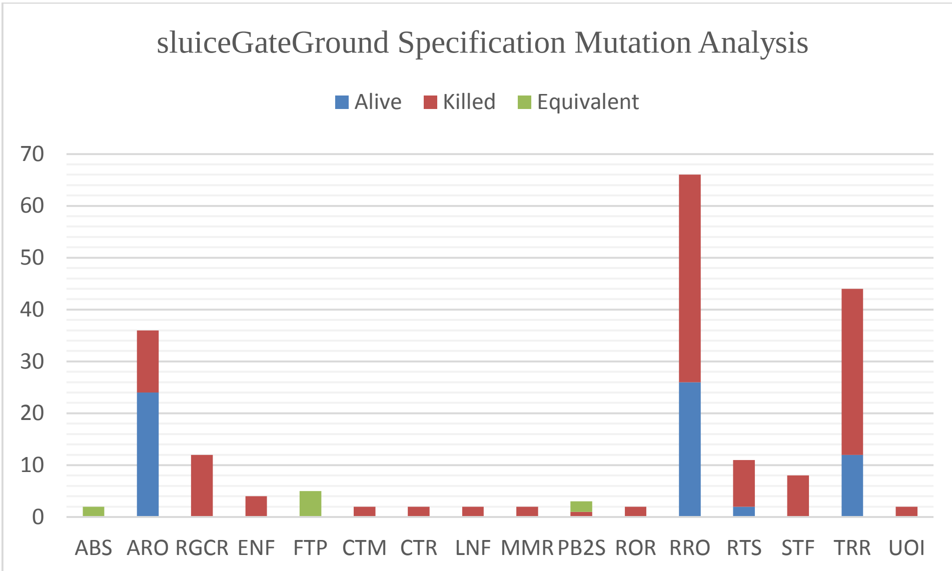


Figure 37: sluiceGateGround specification mutation testing results

6.1.4 Case Study 4: cruiseControl Specification

cruiseControl [113],[114] specification describes an automobile cruise control system. The system consists of engine, ignition, brake pedal, and a cruise control lever. The ignition and engine states could be either in ON or OFF mode. The modes of the cruise control are OFF, INACTIVB (whenever ignition is on, but cruise control is not), CRUISE, and OVERRIDB (whenever cruise control mode is on but is not controlling the speed). The system's conditions indicate whether the ignition is on, the engine is running, the automobile is travelling too fast to be controlled, the brake pedal is being pressed, and whether the cruise control lever is set at Activate, Deactivate, or Resume. The system starts in mode OFF and the cruise control lever is Deactivate.

Cruise control AsmetaL specification has 2 enum domains, 6 functions (a controlled, and 5 monitored), and a macro rule declaration. In addition, it contains 2 invariant definitions. Using MuAsmetaL tool, 421 valid mutants were automatically generated. The generated mutant were run against 102 test cases created using ATGT tool.

Table 29: cruiseControl specification mutation testing results

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	13	7	6	0	46%	**PB2S**	1	0	1	0	100%
ICR	18	0	18	0	100%	**ROR**	8	0	8	0	100%
RGCR	72	0	72	0	100%	**RRO**	48	12	36	0	75%
IDD	2	0	2	0	100%	**RTS**	16	0	16	0	100%
ENF	10	0	10	0	100%	**SBSDL**	4	0	4	0	100%
ERR	36	8	28	0	78%	**STF**	20	0	20	0	100%
FTP	17	0	0	17	*	**TRR**	96	31	65	0	68%
LNF	16	2	14	0	88%	**CTR**	4	2	2	0	50%
LOR	40	1	39	0	98%	**Total**	421	63	341	17	84%

The resulted MS of applying mutation testing on cruiseControl specification is 84% as shown in Table 29. Figure 38 provides a visual illustration of the results.

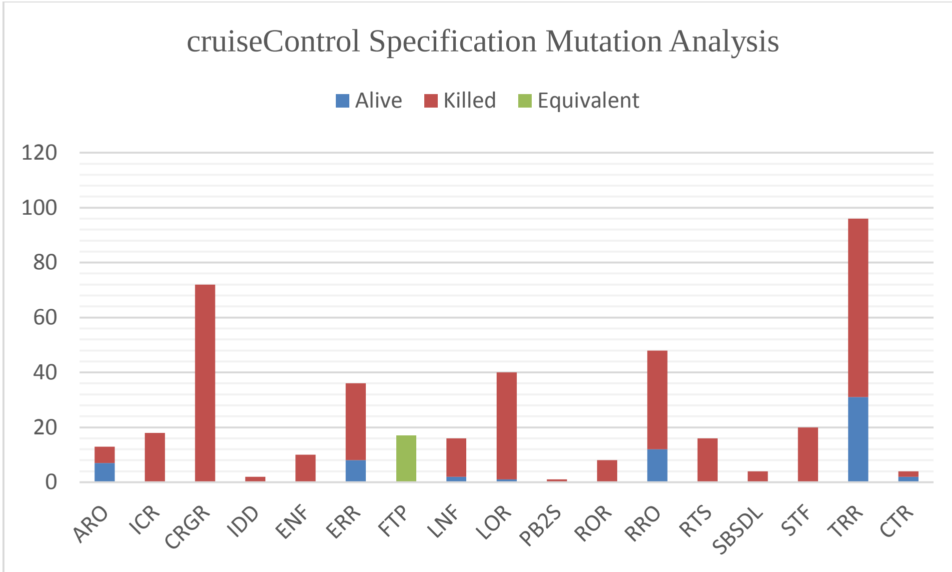


Figure 38: cruiseControl specification mutation testing results

6.1.5 Case Study 5: AdvancedClock Specification

AdvancedClock [115] specification consists of seconds, minutes, and hours. In addition, it continuously increments the seconds by one in each state and recalculated hours: minutes: seconds schema to the correct form.

AdvancedClock AsmetaL specification has 3 sub domains, 3 functions (3 controlled), and 2 macro rule declarations. Using MuAsmetaL tool, 210 valid mutants were automatically generated. The generated mutants were run against only one test case, since there user input is not required, a single run is sufficient.

Table 30: AdvancedClock specification mutation testing results

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS

ABS	11	0	0	11	*	**MMR**	1	0	1	0	100%
AOR	6	2	4	0	67%	**PB2S**	2	0	0	2	*
ARO	13	8	5	0	38%	**ROR**	10	3	7	0	70%
RGCR	2	0	2	0	100%	**RRO**	22	6	16	0	73%
ENF	2	0	2	0	100%	**RTS**	8	2	6	0	75%
FTP	6	0	0	6	*	**STF**	6	2	4	0	67%
CTM	16	10	6	0	38%	**TRR**	11	8	3	0	27%
CTR	80	43	37	0	46%	**UOI**	14	11	3	0	21%
						Total	210	95	96	19	55%

Table 30 shows MS of 55% resulting from applying mutation testing to the AdvancedClock specification. Figure 39 provides a visual illustration of mutation testing results.

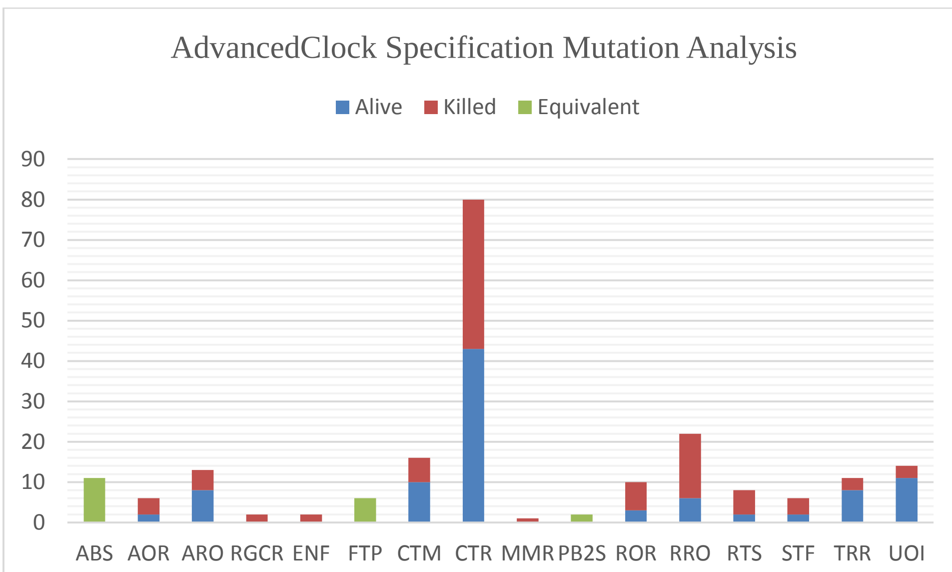


Figure 39: AdvancedClock specification mutation testing results

6.1.6 Case Study 6: AdvancedClock2 Specification

Similar to AdvancedClock specification, AdvancedClock2 [116] consists of seconds, minutes, and hours. However, it increments the time based on user input (*Signal*).

Moreover, the time schema is different from real world, where seconds, minutes, and hours could be {0, 1, 2}.

AdvancedClock2 AsmetaL specification has 3 sub domains, 4 functions (3 controlled, and a monitored), and 2 macro rule declarations. Using MuAsmetaL tool, 204 valid mutants were automatically generated. The generated mutant were run against 45 test cases generated using ATGT Tool.

Table 31: AdvancedClock2 specification mutation testing results

Operator	T	A	F	Eq	MS	Operator	T	A	K	Eq	MS
ABS	11	0	0	11	*	**MMR**	1	0	1	0	100%
ARO	20	8	12	0	60%	**PB2S**	2	2	0	0	0%
RGCR	6	0	6	0	100%	**ROR**	9	2	7	0	78%
ENF	3	0	3	0	100%	**RRO**	24	6	18	0	75%
FTP	9	0	0	9	*	**RTS**	9	0	9	0	100%
CTM	15	5	10	0	67%	**STF**	6	0	6	0	100%
CTR	52	14	38	0	73%	**TRR**	20	7	13	0	65%
LNF	2	1	1	0	50%	**UOI**	15	9	6	0	40%
						Total	204	54	130	20	71%

Table 31 shows the 71% MS resulted from applying mutation testing to AdvancedClock2 specification. Figure 40 provides a visual illustration of mutation testing results.

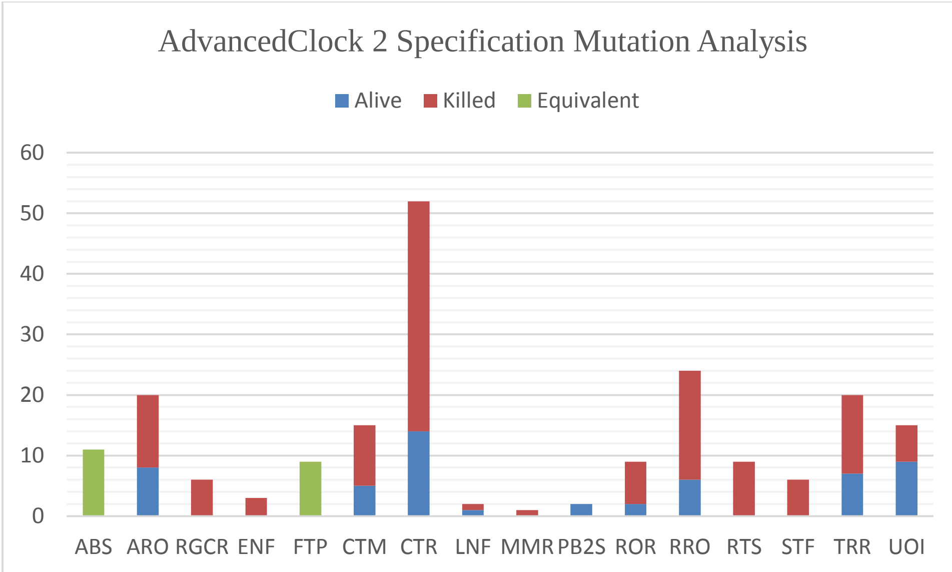


Figure 40: AdvancedClock2 specification mutation testing results

6.1.7 Case Study 7: fattoriale Specification

Fattoriale [117] specification is an implementation of factorial function in AsmetaL Language according to the following equation.

$$n = \begin{cases} 1 & if\ n = 0 \\ (n-1)! \times n & if\ n > 0 \end{cases}$$

It has 4 functions (3 controlled and a monitored), and 2 macro rule definitions. MuAsmetaL generates 136 valid mutants, where ATGT generates 44 test cases.

Table 32: fattoriale specification mutation testing results

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ABS	13	0	0	13	*	**CTR**	6	0	6	0	100%
AOR	4	0	4	0	100%	**LNF**	6	0	6	0	100%
ARO	37	14	23	0	62%	**MMR**	1	0	1	0	100%
RGCR	6	0	6	0	100%	**PB2S**	2	2	0	0	0%
ENF	3	0	3	0	100%	**ROR**	15	3	12	0	80%
ERR	12	3	9	0	75%	**RRO**	24	2	22	0	92%

FTP	9	0	0	9	*	**SSM**	1	0	1	0	100%
CTM	6	0	6	0	100%	**Total**	136	24	90	22	79%

The resulted MS of fattoriale specification is 79% as shown in Table 32. Figure 41 visualizes that results based on status of each mutant per operator.

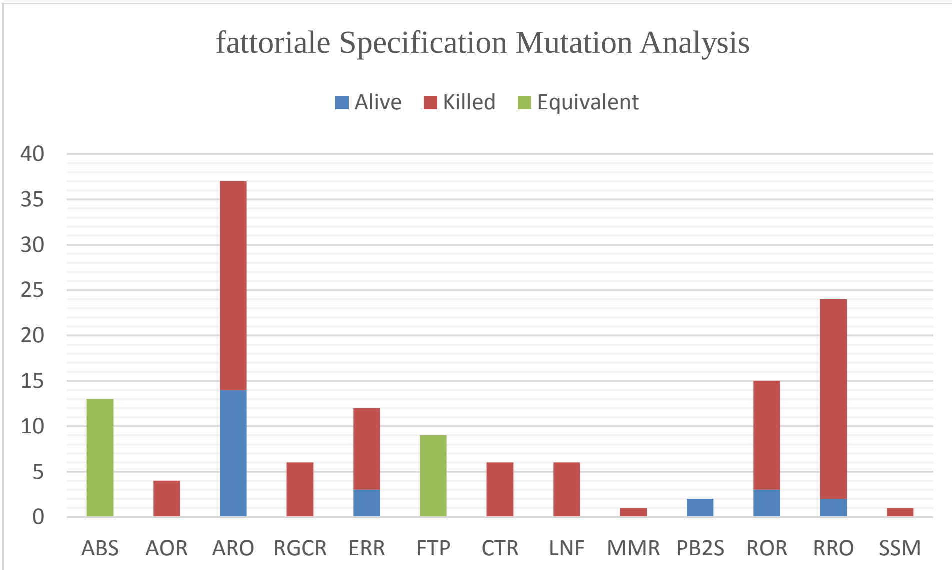


Figure 41: fattoriale specification mutation testing results

6.2 ATGT Test Criteria Comparison using Mutation Testing

We apply our proposed approach over three case studies to assist the adequacy of each test suit generated by different criteria using ATGT (*see section 4.3*).

6.2.1 CruiseControl Specification

Update Rule Coverage (7 TCs)

Table 33: CruiseControl specification mutation testing based on update rule coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	13	8	5	0	38%	**PB2S**	1	1	0	0	0%
ICR	18	0	18	0	100%	**ROR**	8	0	8	0	100%

RGCR	72	6	66	0	92%	**RRO**	48	17	31	0	65%
IDD	2	0	2	0	100%	**RTS**	16	2	14	0	88%
ENF	10	0	10	0	100%	**SBSDL**	4	1	3	0	75%
ERR	36	8	28	0	78%	**STF**	20	2	18	0	90%
FTP	17	0	0	17	*	**TRR**	96	39	57	0	59%
LNF	16	2	14	0	88%	**CTR**	4	2	2	0	50%
LOR	40	5	35	0	88%	**Total**	421	93	311	17	77%

Basic Rule Coverage (12 TCs)

Table 34: CruiseControl specification mutation testing based basic rule coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	13	7	6	0	46%	**PB2S**	1	1	0	0	0%
ICR	18	0	18	0	100%	**ROR**	8	2	6	0	75%
RGCR	72	12	60	0	83%	**RRO**	48	15	33	0	69%
IDD	2	0	2	0	100%	**RTS**	16	3	13	0	81%
ENF	10	0	10	0	100%	**SBSDL**	4	1	3	0	75%
ERR	36	9	27	0	75%	**STF**	20	3	17	0	85%
FTP	17	0	0	17	*	**TRR**	96	49	47	0	49%
LNF	16	5	11	0	67%	**CTR**	4	2	2	0	50%
LOR	40	6	34	0	85%	**Total**	421	115	289	17	72%

MCDC Coverage (32 TCs)

Table 35: CruiseControl specification mutation testing based MCDC coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	13	7	6	0	46%	**PB2S**	1	0	1	0	100%
ICR	18	0	0	0	100%	**ROR**	8	2	6	0	75%
RGCR	72	21	51	0	71%	**RRO**	48	18	30	0	63%
IDD	2	0	0	0	100%	**RTS**	16	7	9	0	56%
ENF	10	0	10	0	100%	**SBSDL**	4	2	2	0	50%
ERR	36	12	24	0	67%	**STF**	20	4	16	0	80%
FTP	17	0	0	17	*	**TRR**	96	57	39	0	41%
LNF	16	4	12	0	75%	**CTR**	4	2	2	0	50%
LOR	40	6	34	0	85%	**Total**	421	142	242	17	60%

Fault Coverage (3 TCs)

Table 36: CruiseControl specification mutation testing based fault coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS

ARO	13	13	0	0	0%	**PB2S**	1	1	0	0	0%
ICR	18	5	13	0	72%	**ROR**	8	8	0	0	0%
RGCR	72	66	6	0	8%	**RRO**	48	48	0	0	0%
IDD	2	1	1	0	50%	**RTS**	16	14	2	0	13%
ENF	10	7	3	0	30%	**SBSDL**	4	4	0	0	0%
ERR	36	36	0	0	0%	**STF**	20	16	4	0	20%
FTP	17	0	0	17	*	**TRR**	96	85	11	0	11%
LNF	16	10	6	0	38%	**CTR**	4	2	2	0	50%
LOR	40	33	7	0	18%	**Total**	421	349	55	17	14%

Pair-wise Coverage (48 TCs)

Table 37: CruiseControl specification mutation testing based pair-wise coverage

Operator	**T**	**A**	**K**	**Eq**	**MS**	**Operator**	**T**	**A**	**K**	**Eq**	**MS**
ARO	13	7	6	0	46%	**PB2S**	1	1	0	0	0%
ICR	18	0	18	0	100%	**ROR**	8	0	8	0	100%
RGCR	72	0	72	0	100%	**RRO**	48	12	36	0	75%
IDD	2	0	2	0	100%	**RTS**	16	0	16	0	100%
ENF	10	0	10	0	100%	**SBSDL**	4	0	4	0	100%
ERR	36	8	28	0	78%	**STF**	20	0	20	0	100%
FTP	17	0	0	17	*	**TRR**	96	31	65	0	68%
LNF	16	2	14	0	88%	**CTR**	4	2	2	0	50%
LOR	40	1	39	0	98%	**Total**	421	64	340	17	84%

The best mutation score is achieved by pair-wise coverage (84%), while, the worst is achieved by fault coverage (14%).

Figure 42 illustrates the results of the application of our proposed approach over CruiseControl specification and overlap between different test case generation criteria by ATGT.

alive mutants.

killed mutants.

killed mutants that cause runtime exceptions.

equivalent mutants

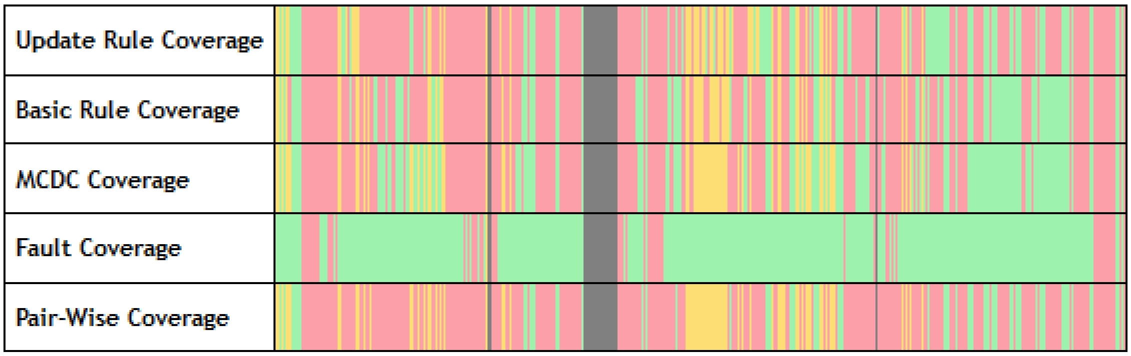


Figure 42: Overall deference of mutation testing over different testing criteria for CruiseControl Specification

6.2.2 RailroadGate Specification

Update Rule Coverage (4 TCs)

Table 38: RailroadGate specification mutation testing based update rule coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	7	4	3	0	43%	LNF	4	0	4	0	100%
ICR	12	2	10	0	83%	PB2S	1	0	1	0	100%
RGCR	8	0	8	0	100%	ROR	19	7	12	0	63%
IDD	3	1	2	0	67%	RRO	12	4	8	0	67%
ENF	5	0	5	0	100%	RTS	7	1	6	0	86%
ERR	12	6	6	0	50%	LOR	60	24	36	0	60%
CTR	3	0	3	0	100%	STF	10	1	9	0	90%
FTP	18	0	0	18	*	TRR	12	4	8	0	67%
						Total	193	54	121	18	69%

Basic Rule Coverage (3 TCs)

Table 39: RailroadGate specification mutation testing based basic rule coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	7	4	3	0	43%	LNF	4	0	4	0	100%
ICR	12	2	10	0	83%	PB2S	1	0	1	0	100%
RGCR	8	0	8	0	100%	ROR	19	8	11	0	58%
IDD	3	1	2	0	67%	RRO	12	4	8	0	67%
ENF	5	0	5	0	100%	RTS	7	2	5	0	71%
ERR	12	6	6	0	50%	LOR	60	27	33	0	55%
CTR	3	0	3	0	100%	STF	10	1	9	0	90%
FTP	18	0	0	18	*	TRR	12	6	6	0	50%
						Total	193	61	114	18	65%

MCDC Coverage (26 TCs)

Table 40: RailroadGate specification mutation testing based basic MCDC coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	7	4	3	0	43%	LNF	4	0	4	0	100%
ICR	12	2	10	0	83%	PB2S	1	0	0	0	100%
RGCR	8	0	8	0	100%	ROR	19	6	13	0	68%
IDD	3	1	2	0	67%	RRO	12	4	8	0	67%
ENF	5	0	5	0	100%	RTS	7	1	6	0	86%
ERR	12	5	7	0	58%	LOR	60	21	39	0	65%
CTR	3	0	3	0	100%	STF	10	1	9	0	90%

FTP	18	0	0	18	*	**TRR**	12	4	8	0	67%
						Total	193	49	125	18	71%

Fault Coverage (8 TCs)

Table 41: RailroadGate specification mutation testing based fault coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	7	4	3	0	43%	**LNF**	4	0	4	0	100%
ICR	12	2	10	0	83%	**PB2S**	1	0	1	0	100%
RGCR	8	0	8	0	100%	**ROR**	19	7	12	0	63%
IDD	3	1	2	0	67%	**RRO**	12	4	8	0	67%
ENF	5	0	5	0	100%	**RTS**	7	1	6	0	86%
ERR	12	5	7	0	58%	**LOR**	60	22	38	0	63%
CTR	3	0	3	0	100%	**STF**	10	1	9	0	90%
FTP	18	0	0	18	*	**TRR**	12	4	8	0	67%
						Total	193	51	124	18	71%

Pair-wise Coverage (20 TCs)

Table 42: RailroadGate specification mutation testing based pair-wise coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	7	4	3	0	43%	**LNF**	4	0	4	0	100%
ICR	12	2	10	0	83%	**PB2S**	1	0	1	0	100%
RGCR	8	0	8	0	100%	**ROR**	19	3	16	0	84%
IDD	3	1	2	0	67%	**RRO**	12	4	8	0	67%
ENF	5	0	5	0	100%	**RTS**	7	1	6	0	86%
ERR	12	5	7	0	58%	**LOR**	60	11	49	0	82%
CTR	3	0	3	0	100%	**STF**	10	1	9	0	90%
FTP	18	0	0	18	*	**TRR**	12	4	8	0	67%
						Total	193	36	139	18	79%

Three-wise Coverage (16 TCs)

Table 43: RailroadGate specification mutation testing based three-wise coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ARO	7	4	3	0	43%	**LNF**	4	0	4	0	100%
ICR	12	2	10	0	83%	**PB2S**	1	0	1	0	100%
RGCR	8	0	8	0	100%	**ROR**	19	4	15	0	79%
IDD	3	1	2	0	67%	**RRO**	12	4	8	0	67%
ENF	5	0	5	0	100%	**RTS**	7	1	6	0	86%

ERR	12	5	7	0	58%	**LOR**	60	15	45	0	75%
BTR	3	0	3	0	100%	**STF**	10	1	9	0	90%
FTP	18	0	0	18	*	**TRR**	12	4	8	0	67%
						Total	193	41	134	18	77%

The best mutation score is achieved by pair-wise coverage (86%), while, the worst is achieved by basic rule coverage (65%).

Figure 43 illustrates the results of the application of our proposed approach over RailroadGate specification and overlap between different test case generation criteria by ATGT.

alive mutants.

killed mutants.

killed mutants that cause runtime exceptions.

equivalent mutants.

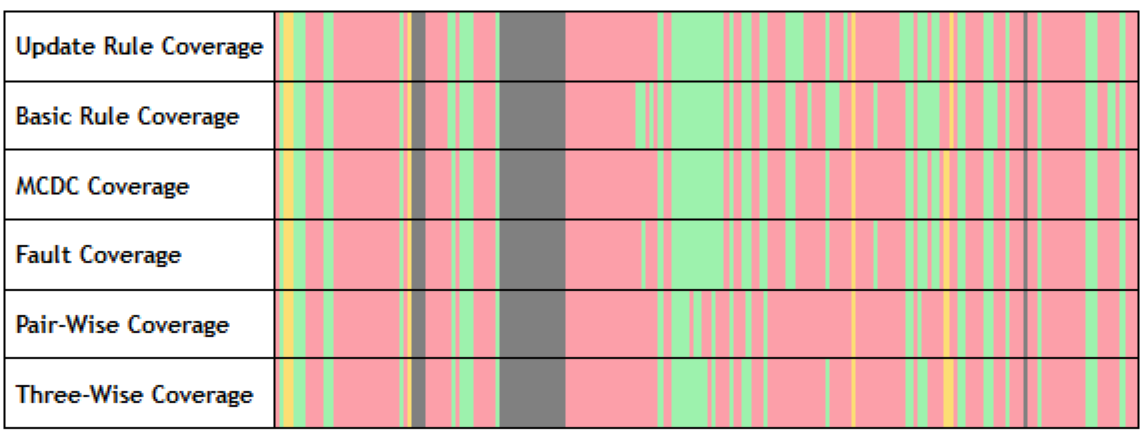


Figure 43: Overall deference of mutation testing over different testing criteria for RailroadGate Specification

6.2.3 SluiceGateGround Specification

Update Rule Coverage (2 TCs)

Table 44: SluiceGateGround specification mutation testing based update rule coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ABS	2	0	0	2	*	MMR	2	0	2	0	100%
ARO	36	27	9	0	25%	PB2S	3	1	0	2	0%
RGCR	12	1	11	0	92%	ROR	2	0	2	0	100%
ENF	4	0	4	0	100%	RRO	66	29	37	0	56%
FTP	5	0	0	5	*	RTS	11	2	9	0	82%
CTM	2	0	2	0	100%	STF	8	1	7	0	88%
CTR	2	0	2	0	100%	TRR	44	12	32	0	73
LNF	2	0	2	0	100%	UOI	2	0	2	0	100%
						Total	203	73	121	9	62%

Basic Rule Coverage (6 TCs)

Table 45: SluiceGateGround specification mutation testing based basic rule coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ABS	2	0	0	2	*	MMR	2	0	2	0	100%
ARO	36	24	12	0	33%	PB2S	3	0	1	2	100%
RGCR	12	1	11	0	92%	ROR	2	0	2	0	100%
ENF	4	0	4	0	100%	RRO	66	26	40	0	61%
FTP	5	0	0	5	*	RTS	11	2	9	0	82%
CTM	2	0	2	0	100%	STF	8	0	8	0	100%
CTR	2	0	2	0	100%	TRR	44	12	32	0	73%
LNF	2	0	2	0	100%	UOI	2	0	2	0	100%
						Total	203	65	129	9	66%

MCDC Coverage (8 TCs)

Table 46: SluiceGateGround specification mutation testing based MCDC coverage

Operator	T	A	K	Eq	MS	Operator	T	A	K	Eq	MS
ABS	2	0	0	2	*	MMR	2	0	2	0	100%
ARO	36	24	12	0	33%	PB2S	3	0	1	2	100%
RGCR	12	0	12	0	100%	ROR	2	0	2	0	100%
ENF	4	0	4	0	100%	RRO	66	26	40	0	61%
FTP	5	0	0	5	*	RTS	11	2	9	0	82%
CTM	2	0	2	0	100%	STF	8	0	8	0	100%

CTR	2	0	2	0	100%	**TRR**	44	12	32	0	73%
LNF	2	0	2	0	100%	**UOI**	2	0	2	0	100%
						Total	203	64	130	9	67%

Fault Coverage (26 TCs)

Table 47: SluiceGateGround specification mutation testing based fault coverage

Operator	**T**	**A**	**K**	**Eq**	**MS**	**Operator**	**T**	**A**	**K**	**Eq**	**MS**
ABS	2	0	0	2	*	**MMR**	2	0	2	0	100%
ARO	36	24	12	0	33%	**PB2S**	3	0	1	2	100%
RGCR	12	0	12	0	100%	**ROR**	2	0	2	0	100%
ENF	4	0	4	0	100%	**RRO**	66	26	40	0	61%
FTP	5	0	0	5	*	**RTS**	11	2	9	0	82%
CTM	2	0	2	0	100%	**STF**	8	0	8	0	100%
CTR	2	0	2	0	100%	**TRR**	44	12	32	0	73%
LNF	2	0	2	0	100%	**UOI**	2	0	2	0	100%
						Total	203	64	130	9	67%

Pair-wise Coverage (4 TCs)

Table 48: SluiceGateGround specification mutation testing based pair-wise coverage

Operator	**T**	**A**	**K**	**Eq**	**MS**	**Operator**	**T**	**A**	**K**	**Eq**	**MS**
ABS	2	0	0	2	*	**MMR**	2	0	2	0	100%
ARO	36	24	12	0	33%	**PB2S**	3	0	1	2	100%
RGCR	12	0	12	0	100%	**ROR**	2	0	2	0	100%
ENF	4	0	4	0	100%	**RRO**	66	26	40	0	61%
FTP	5	0	0	5	*	**RTS**	11	2	9	0	82%
CTM	2	0	2	0	100%	**STF**	8	0	8	0	100%
CTR	2	0	2	0	100%	**TRR**	44	12	32	0	73%
LNF	2	0	2	0	100%	**UOI**	2	0	2	0	100%
						Total	203	64	130	9	67%

The best mutation score is achieved by MCDC coverage, fault coverage, and pair-wise coverage (67%), while, the worst is achieved by update rule coverage (62%).

Figure 44 illustrates the results of the application of our proposed approach over SluiceGateGround specification and overlap between different test case generation criteria by ATGT.

alive mutants.

killed mutants.

killed mutants that cause runtime exceptions.

equivalent mutants.

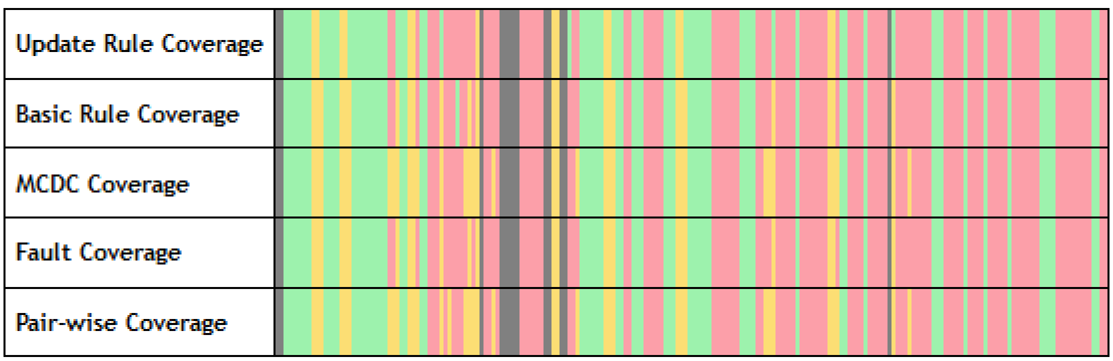


Figure 44: Overall deference of mutation testing over different testing criteria for SluiceGateGround Specification

6.2.4 Results Summary

Based on the above empirical evaluation, the test cases generated by ATGT vary in their ability to kill certain mutants.

	CruiseControl	RailroadGate	SluiceGateGround	Average
Update Rule Coverage	77%	69%	62%	69%

Basic Rule Coverage	72%	65%	66%	68%
MCDC Coverage	60%	71%	67%	66%
Fault Coverage	14%	71%	67%	51%
Pair-wise Coverage	84%	79%	67%	77%
Three-wise Coverage	N/A	77%	N/A	N/A

Based on the achieved mutation score for the three different case studies, we can conclude observed that the best mutation score is achieved the pair wise test coverage criteria (average mutation score of 77% based on the table above). In addition, update rule coverage and basic rule coverage mutation scores are close, however, update rule coverage is slightly better and that is conformed to coverage strength order in section 3.1.3.

6.3 Chapter Summary

In this chapter, an empirical investigation is performed on seven case studies, in order to evaluate the proposed set of AsmetaL mutation operators. The proposed set of operators for AsmetaL is able to generate syntactically correct AsmetaL mutants. In addition, it is observed that different test cases for the case studies vary in their ability to kill mutants. Therefore, it is possible to rely on the proposed set of AsmetaL mutation operators in assessing and comparing the performance of different test cases. The goals of the application mutation testing are achieved by the proposed mutation operator for AsmetaL specification. In addition, as an application of the proposed approach, different test case generation criteria provided by ATGT are evaluated based on the achieved mutation score.

CHAPTER 7

Application of Cost Reduction Techniques to AsmetaL Mutation Testing

7.1 Introduction

Mutation testing has proven its effectiveness in detecting inadequacy in the testing suites. However, mutation testing suffers from high computation problem where a few line of code specification may result in over thousand faulty versions (*mutants*) [96]. The high computation cost may hinder the adoption of mutation testing by practitioners. Many techniques have been proposed to reduce the computation cost of mutation testing, such as selective mutation (2-selective, 4-selective, 6-selective) and random mutation.

Gligoric et al.[98] have investigated the application of selective mutation on concurrent operators, the conclusion of their study is that operator-based selection preformed slightly better than random-based selection. However, Zhang et al.[99] have conducted a study on comparing the application of operator-based selection verses random based selection with respect to the resulting effectiveness and cost saving. Their work was conducted in the context of the C programming language and they have shown that random-based selection is superior to all types of operator-based selection. In addition, Zhang et al.[100] have proposed a technique in which it combines operator-based and random-based to achieve better results. Their approach is based on four strategies i) Baseline: selects x% mutants from a selected set. ii) MOp-Based: selects x% mutants produces by each operators. iii)

PElem-Based: selects x% mutants produced by mutating the same program element. iv) PElem-Mop-based: selects x% mutants produces by each operators by mutating the same program element. Their approach resulted in 95% mutant's reduction while reducing the execution cost by 93.46%.

Mresa et al.[101] have evaluated the efficiency of mutation operators by the ratio of mutation score to the cost of mutation testing. Zhang et al.[102] investigated the reduction of cost by applying test prioritization inspired by regression test case prioritization technique to an effective testing sequence. Namin et al.[103][104] have proposed an approach for selecting a sufficient set of mutants based on several criteria of statistical analysis including all-subsets regression, elimination-based correlation technique, and cluster analysis.

The basic idea behind selective mutation analysis is that killing a mutant may lead to killing other mutants as well. Thus, running test suite against a set of selected mutants might be considered sufficient as substitution of the full set. In this chapter, we have investigated applying selective mutation operator-based and random-based in order to demonstrate the tradeoff between effectiveness and saving. Moreover, we have investigated the relationship between operator-based and random-based mutation. Although, Zhang et al.[99] have concluded that all operator-based selection are not superior to random-based selection.

In order to compare operator-based and random-based in the AsmetaL context, we adopt a set of questions introduced in [55][106][96][98].

Q1[56]: What are the most dominant mutation operators out of the proposed AsmetaL mutation operators?

Q2[55][56]: Is N-selective mutation applicable in the context AsmetaL?

Q3: Is random-based selective mutation applicable in the context Abstract State Machines?

Q4[98]: How do operator-based and random-based mutant selection compare in the context of AsmetaL?

Q5[99]: Does random-based mutant selection provide a stable mutation scores in the context of AsmetaL?

7.2 Evaluation Criteria of the Mutation Operators Cost Reduction Techniques

In what follows, we present the criteria used to evaluate the application of the cost reduction techniques to mutation testing.

7.2.1 Effectiveness

In order to acquire the level of effectiveness of applying selective mutation, we have formulated the problem as follows. Given a specification (denoted as S) and a set of mutants (denoted as M) generated for S by applying all mutation operators, equivalent mutants are removed from M and a set of non-equivalent mutants (denoted as M_{nq}) is acquired. After applying all test cases generated by ATGT tool, all non-killable (*alive*) mutants in M are considered as equivalent mutants, as done by previous studies [55][56][98][99][103] and removed. A reduced set of test cases (denoted as T) is considered as M_{nq} sufficient, if for any mutant in $M_{nq,}$ there is at least one test case that is able to kill it. Similarly, a reduced set of test cases (denoted as $T_S \subseteq T$) and a set of mutants (denoted as $M_S \subseteq M_{nq}$). T_S is said to be M_S sufficient, if for any mutant in $M_{S,}$ there is at least one test case in T_S that is able to kill it. The mutation score presented by applying T_S

against all mutants M_{nq}, represents the effectiveness of applying selective set of mutants M_S.

$$Eff(M_S, M_{nq}) = MS(T_S, M_{nq})$$

Figure 45 shows the procedure of acquiring the effectiveness of a set of selective mutants.

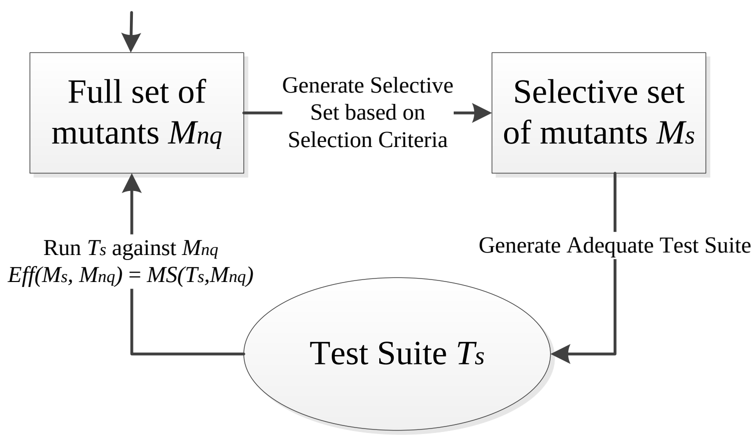


Figure 45: Selective mutation reduction procedure

7.2.2 Cost Saving

Saving is acquired as a difference between the execution cost of running the set of all mutants and the execution cost of running selective set of mutants normalized by the execution cost of running the set of all mutants. Originally, Offutt [56] has considered the cost in term of number of generated mutants. However, the number of generated mutants is not a precise indicator to the actual cost of preforming mutation testing.

$$Saving\left(M_S, M_{nq}\right) = 1 - \frac{Cost(M_S)}{Cost(M_{nq})}$$

Where Cost(Ms) denotes the cost of running the set of selected mutants. And Cost (Mnq) denotes the cost of running all none equivalent mutants.

Mresa et al.[101] proposed that the execution cost is acquired by counting the exact number of execution of test cases against mutants.

$$execCnt(t, M) = \#M$$
$$+ \# M - \# kill(M, \{c_1\})$$
$$+ \# M - \# kill(M, \{c_1, c_2\}) \ ...$$
$$+ \# M - \# kill(M, \{c_1, c_2, \dots, c_{n-1}\})$$

Where #M is the number of mutants. And # $kill(M, \{c_1\})$ represents the number of killed mutants by test case c_1.

In this study we follow Mresa [101] technique to acquire the exact number of execution.

$$Saving\ (M_S, M_{nq}) = 1 - \frac{execCnt(T_S, M_S)}{execCnt(T, M_{nq})}$$

7.2.3 Stability

In the case of the application of the random selection technique, standard deviation can indicate the level of stability in the random sample. Zhang et al.[99] used 50 random runs to calculate the stability (*standard deviation*) of randomly selected samples of mutants for effectiveness and saving. The standard deviation will be calculated based on 100 random runs. The effectiveness and saving are calculated as the average of 100 random runs.

7.3 N-selective-based Mutation

N-selective-based mutation testing is performed by applying all mutation operators to the original specification resulting in a set of mutants, denoted as M. Mutants generated by the N most dominant operators (*dominant in number of generated mutants*) are discarded. The rest of mutants are to be considered the selective set of mutants, denoted as M_S, based on which the effectiveness and saving are drawn to assist the performance of that M_S to M. Based on Offutt [55] work, we have applied 2-selective, 4-selective and 6-selective to set of case studies introduced in Chapter 6.

7.4 Random-based Selective Mutation

Similarly, random-based mutation testing technique acquires a set of mutants, denoted as M, generated by all mutation operators. The set of selected mutants, denoted as M_S, is sample of *x%* size of M by uniformly random distribution. In our study, we have chosen to investigate the level of effectiveness and saving by applying 10%, 25%, 50% random set of mutants.

7.5 Applying Cost Reduction Techniques to Case Studies

7.5.1 Case Study 1: ferrymanSimulator Specification

Operator-based Selection Mutation

2-selective

The two most dominant operators for ferrymanSimulator specification are ETR and RRO, producing 19%, and 15% of the overall mutants. The elimination of mutants they produce will results in 100% effectiveness and 30.14% saving.

4-selective

In addition to ETR and RRO, we expand the set of selected mutants adding CRRO and ARO operators. The overall set of 4 selected operators are producing 52% of mutants. Hence, the level of effectiveness is 98.26%, while the saving is 56.81%.

6-selective

Moreover, the consideration of RTS and TTR (*overall set produces 63% of mutants*) results in 98.26% effectiveness and 68.12% saving.

Random-based Selection Mutation

10% Random-based Selection Mutation

Figure 46 shows 100 runs 10% random selection based mutation testing. The average level of effectiveness is 96.77% while the standard deviation is 0.032. The saving is 87.33% with standard deviation of 0.013.

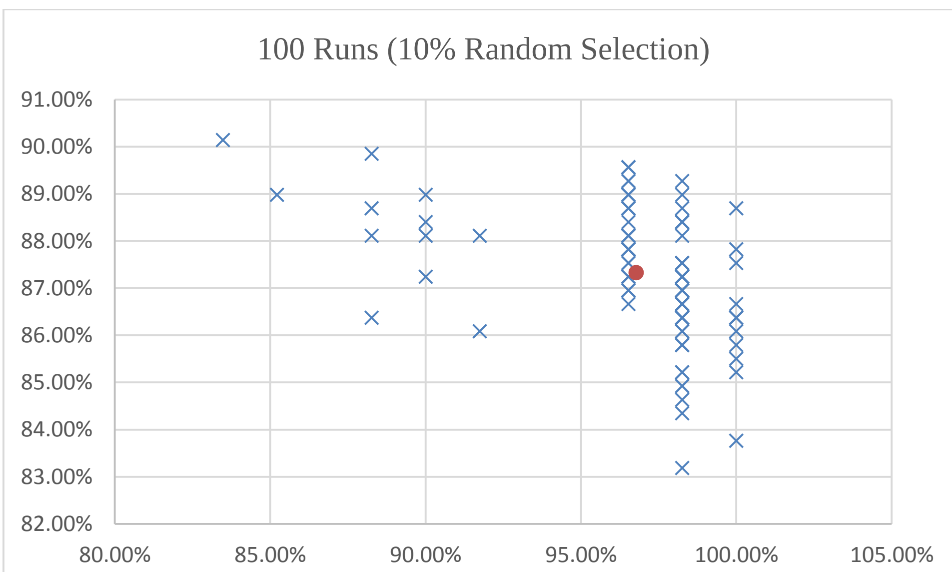


Figure 46: ferrymanSimulator specification random selection (10%)

25% Random-based Selection Mutation

While applying 25% random selection results in 99.08% average effectiveness and average 68.82% saving. The standard deviation is 0.011 and 0.02 respectively. Figure 47 illustrates a 100 random runs with 25% sample size.

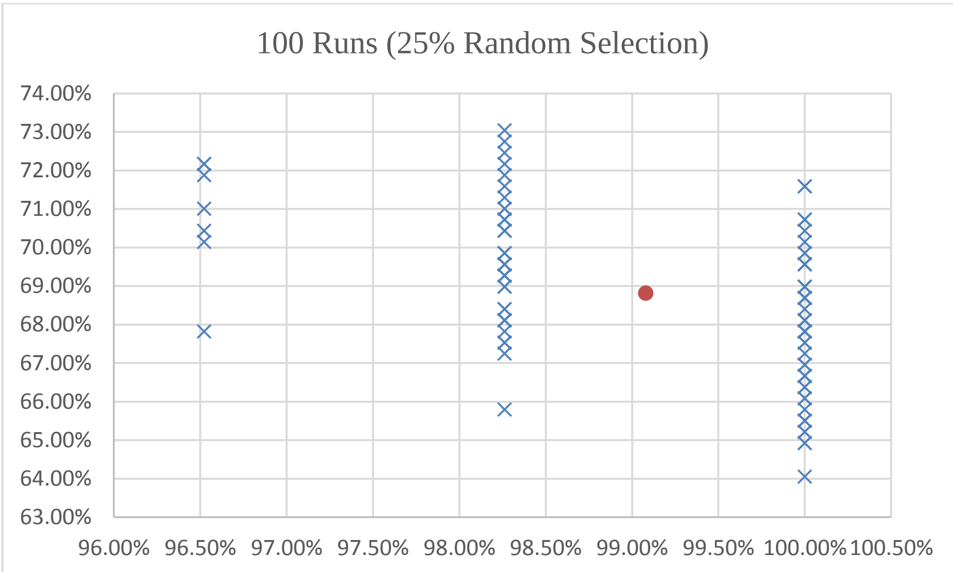


Figure 47: ferrymanSimulator specification random selection (25%)

50% Random-based Selection Mutation

The application of 50% random selection produces 99.91% average of effectiveness and 37.65% average of saving (*Standard deviations are 0.004 and 0.018 respectively*). Figure 48 illustrates a 100 random runs with 50% sample size.

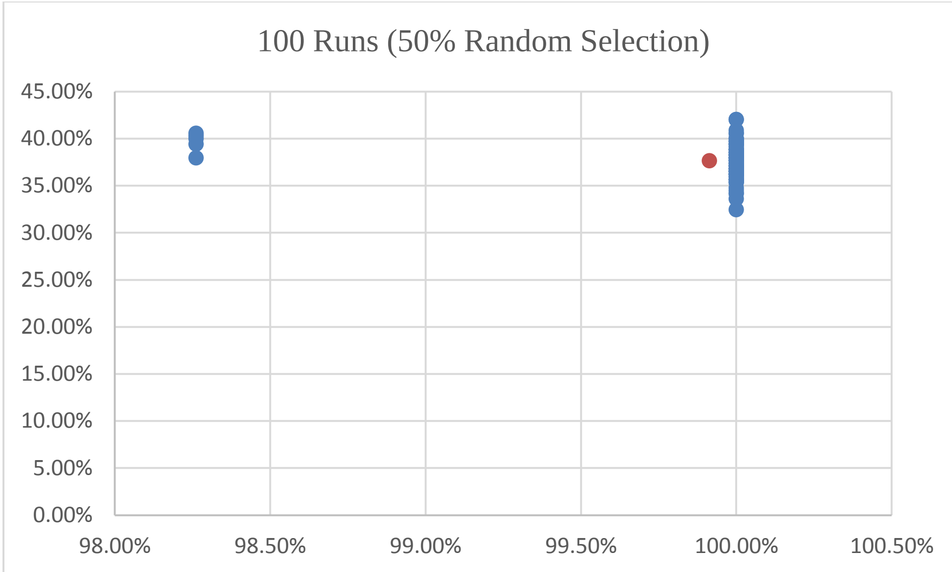


Figure 48: ferrymanSimulator specification random selection (50%)

7.5.2 Case Study 2: railroadGate Specification

Operator-based Selection Mutation

2-selective

Based on number of generated mutants, ROR and ICR are the most dominant operators, in which they produce 14% and 9% respectively. The level of effectiveness maintained while excluding their mutants is 100%. In addition, the level of saving is 19.67%.

4-selective

Introducing two more operators (*ERR and RRO*) to the previous set of selected operators (overall set produces 41% of overall mutants) maintains 100% effeteness and reduces the computation cost by 28.42%.

6-selective

Furthermore, the inclusion of TRR (9%) and STF (8%) to the selected list (58% of mutants) results in 100% effectiveness and 38.25% saving.

Random-based Selection Mutation

10% Random-based Selection Mutation

Applying 10% random selection mutation testing would results on average of 95.07% effectiveness and 85.55% saving. Figure 49 illustrates a 100 random runs over 10% random sample size (standard deviations are 0.049 and 0.018 respectively).

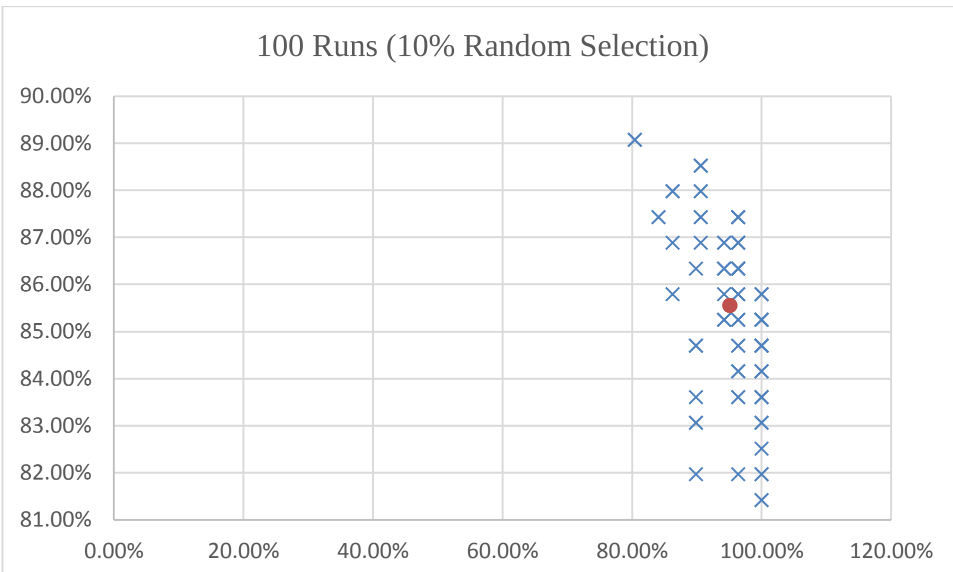


Figure 49: railroadGate specification random selection (10%)

25% Random-based Selection Mutation

As shown in Figure 50, a 100 random runs over 25% random sample size will results on average of 99.62% effectiveness and 64.55% saving (standard deviations are 0.012 and 0.021 respectively).

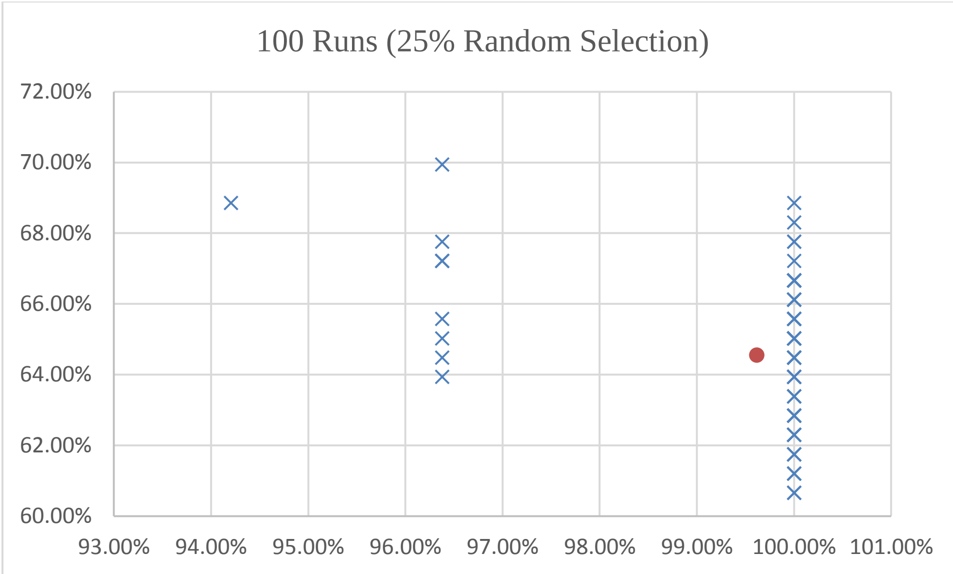


Figure 50: railroadGate specification random selection (25%)

50% Random-based Selection Mutation

The average level of effectiveness is 100% and the average level of saving is 29.9% (standard deviations are 0 and 0.023 respectively). Figure 51 shows a 100 runs with sample size of 50%.

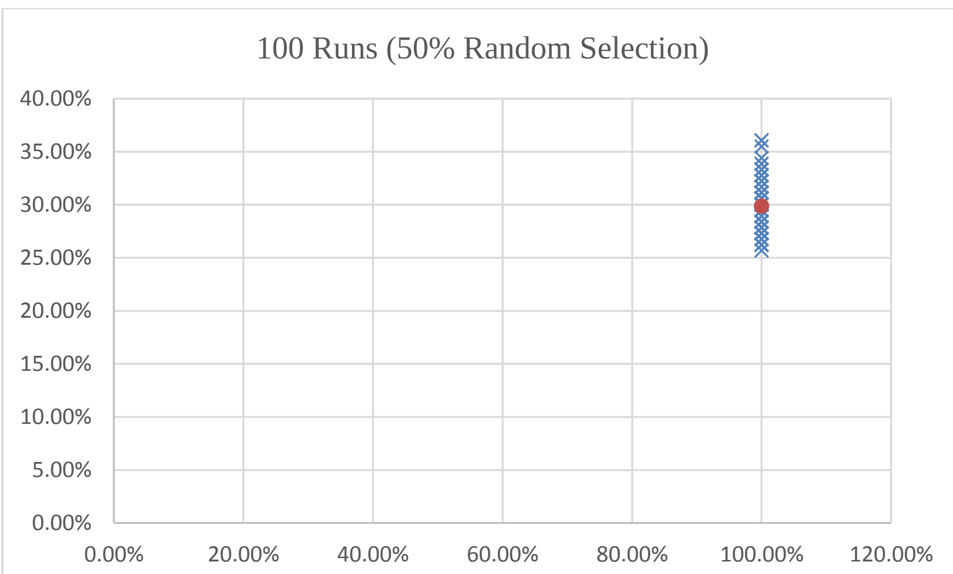


Figure 51: railroadGate specification random selection (50%)

7.5.3 Case Study 3: sluiceGateGround Specification

Operator-based Selection Mutation

2-selective

The set of selected mutation operators includes RRO (*32% of mutants*) and TRR (*21% of mutants*). The acquired effectiveness is 100% while the saving is 55%.

4-selective

The inclusion of ARO (*17% of mutants*) and RGCR (*6% of mutants*) operators to the previous operators (*76% of mutants*) grants 99.24% effectiveness and 75.5% saving.

6-selective

Moreover, including RTS (*5% of mutants*) and STF (*4% of mutants*) operators (*85% of overall mutants*) results in 96.18% effectiveness and 89% saving.

Random-based Selection Mutation

10% Random-based Selection Mutation

The application of 10% random selection results on averages of 96.42% (*standard deviation of 0.020*) effectiveness and 83.93% (*standard deviation of 0.015*). Figure 52 illustrates a 100 runs with random sample of 10% size.

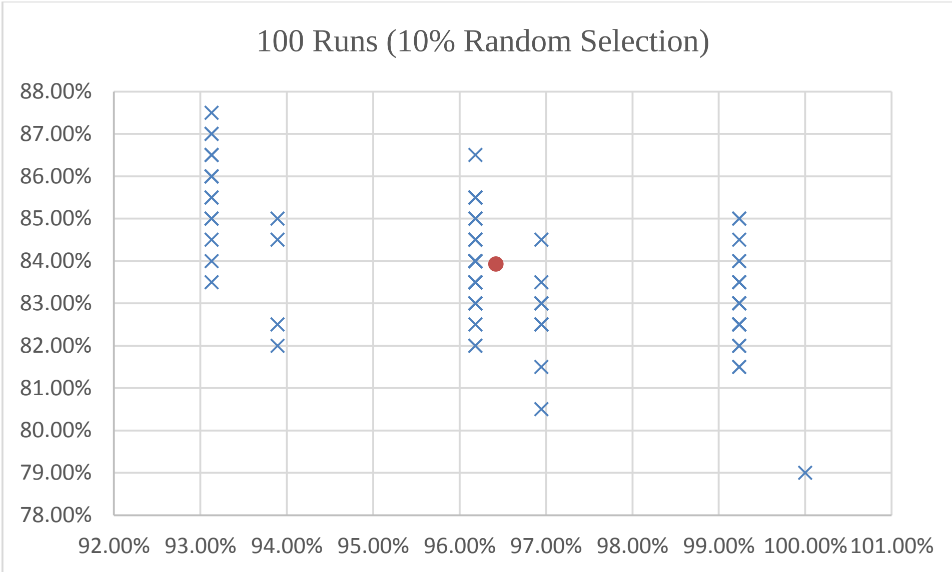


Figure 52: sluiceGateGround specification random selection (10%)

25% Random-based Selection Mutation

While applying 25% random selection results on averages of 98.77% (*standard deviation of 0.013*) effectiveness and 60.13% (*standard deviation of 0.02*) saving. Figure 53 illustrates a 100 runs with random sample of 25% size.

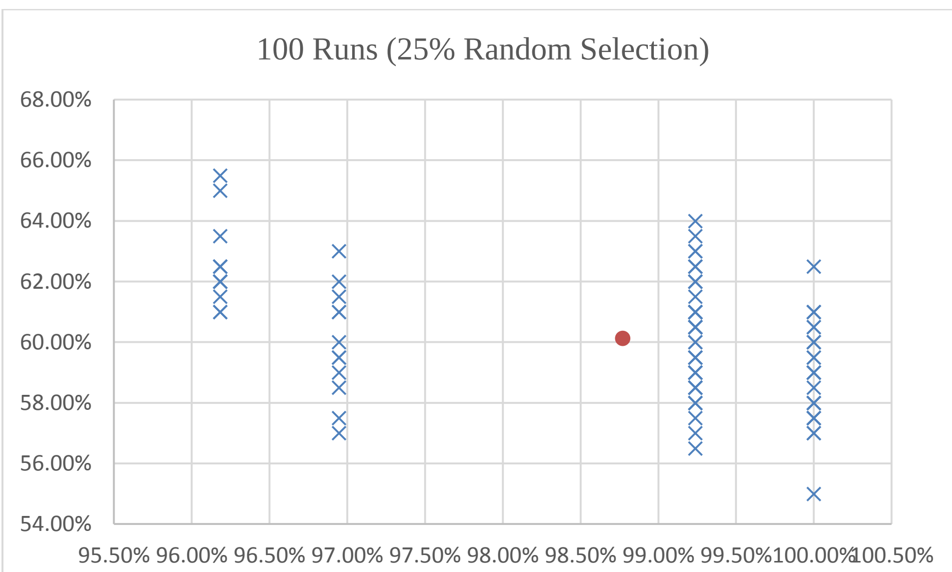


Figure 53: sluiceGateGround specification random selection (25%)

50% Random-based Selection Mutation

Nevertheless, the application of 50% random selection results on averages of 99.84% (*standard deviation of 0.015*) effectiveness and 20.35% (*standard deviation of 0.02*) saving. Figure 54 illustrates a 100 runs with random sample of 50% size.

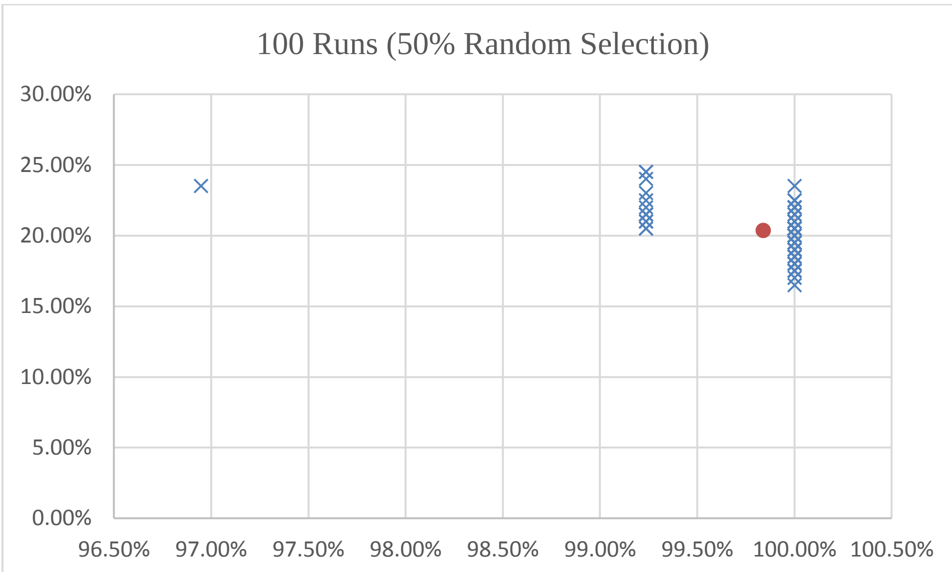


Figure 54: sluiceGateGround specification random selection (50%)

7.5.4 Case Study 4: cruiseControl Specification

Operator-based Selection Mutation

2-selective

TRR (*23% of mutants*) and RGCR (*17% of mutants*) are the most dominant operators for cruiseControl specification. The elimination of mutants generated by them results in 99.12% effectiveness and 42.42% saving.

4-selective

Furthermore, the including of RRO (*11% of mutants*) and LOR (*10% of mutants*) operators to the previous set of selected operators (*overall of 61% of mutants*) results in 98.53% effectiveness and 66.62% saving.

6-selective

ERR and STF produce 9% and 5% of mutants respectively. The set of the six operators is responsible for 74% of overall generated mutants. Similar to 4-selective, the level of effectiveness is 98.53%, however, the level of saving is 80.09%.

Random-based Selection Mutation

10% Random-based Selection Mutation

Figure 55 shows a 100 runs of 10% random selection. The average of effeteness is 95.99% (*standard deviation of 0.021*), while the average of saving is 87.30% (*standard deviation of 0.011*).

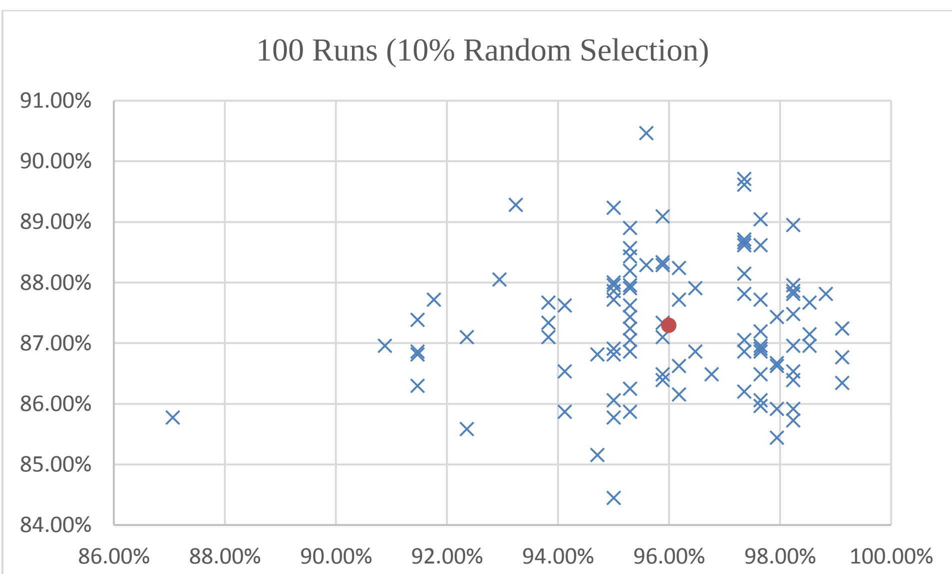


Figure 55: cruiseControl specification random selection (10%)

25% Random-based Selection Mutation

Figure 56 shows a 100 runs of 25% random selection. The average of effeteness is 98.47% (*standard deviation of 0.022*), while the average of saving is 68.83% (*standard deviation of 0.013)*.

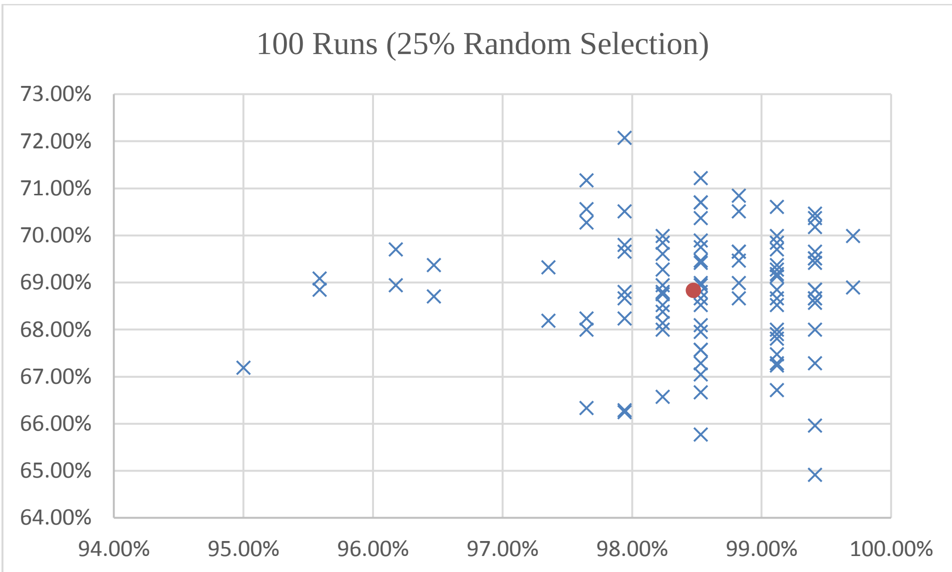


Figure 56: cruiseControl specification random selection (25%)

50% Random-based Selection Mutation

The average of effeteness is 99.48% (standard deviation of 0.004), while the average of saving is 38.13% (standard deviation of 0.016). Figure 57 shows a 100 runs 50% random selection.

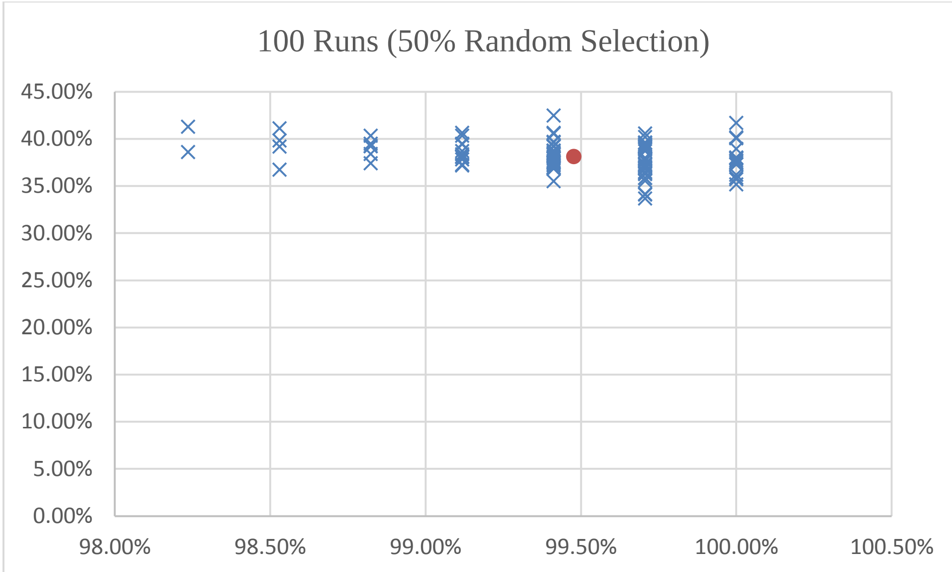


Figure 57: cruiseControl specification random selection (50%)

7.5.5 Case Study 5: AdvancedClock Specification

There is not mutation selection investigation done for this case study, since it only has one test case (*deterministic specification that does not required user input*). Thus, a test case that kills a single mutant would kill them all with 100% effectiveness.

7.5.6 Case Study 6: AdvancedClock2 Specification

There is not mutation selection investigation done for this case study, since only one test case is effective among all test cases generated by ATGT while other test cases do not contribute by killing any mutants therefore, they should be discarded. Thus, only a single test case is kept in the test suite.

7.5.7 Case Study 7: fattoriale Specification

Operator-based Selection Mutation

2-selective

The elimination of the two most dominant operators are ARO (*26% of mutants*) and RRO (*17% of mutants*) results in 98.94% effeteness and 40.58% saving.

4-selective

The introduction of the next two dominant operators ROR (*11% of mutants*) and ABS (*9% of mutants*), the four operators are responsible of 63% of overall mutants, results in 96.81% effeteness and 52.17% saving.

6-selective

Furthermore, the consideration of ERR (*9% of mutants*) and CTR (*4% of mutants*) results in 96.81% effectiveness and 73.91% saving.

Random-based Selection Mutation

10% Random-based Selection Mutation

The average level of effectiveness results from 10% random selection is 93.69% (*standard deviation 0.034*) while the saving is 84.20% (*standard* deviation 0.027). Figure 58 shows a 100 runs of sample with size of 10%.

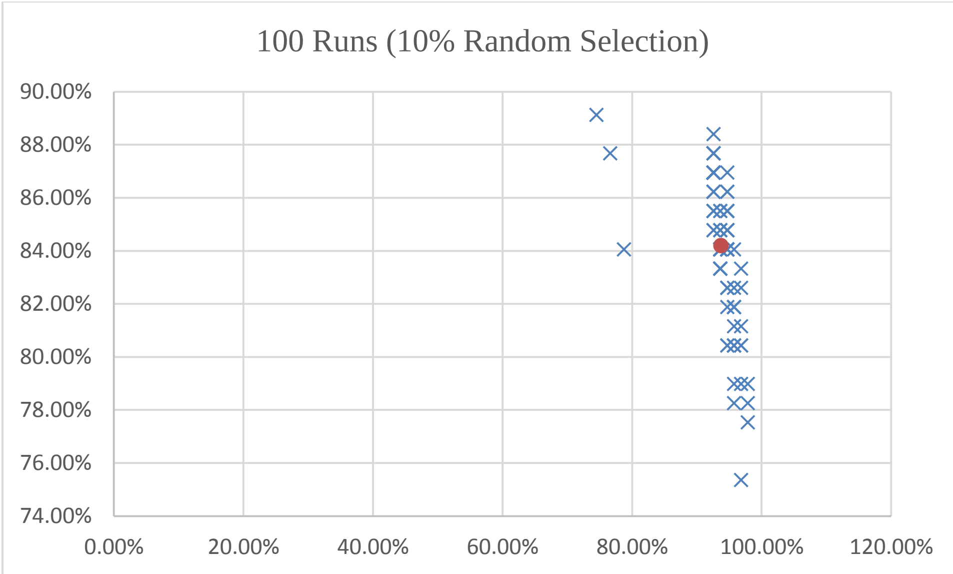


Figure 58: fattoriale specification random selection (10%)

25% Random-based Selection Mutation

The average level of effectiveness results from 25% random selection is 96.44% (*standard deviation 0.016*) while the saving is 62% (*standard deviation 0.034*). Figure 59 shows a 100 runs of sample with size of 25%.

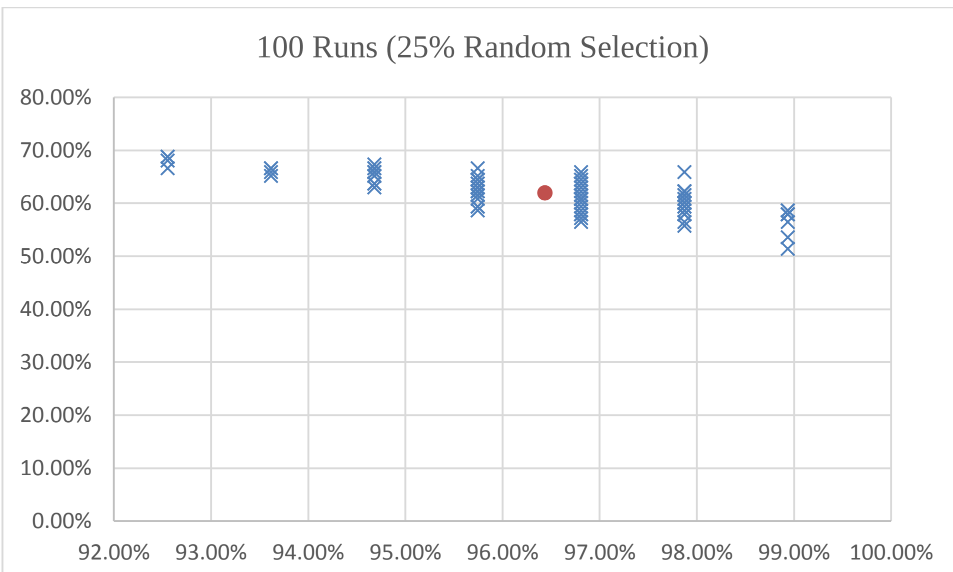


Figure 59: fattoriale specification random selection (25%)

50% Random-based Selection Mutation

The average level of effectiveness results from 50% random selection is 98.90% (*standard deviation 0.0096*) while the saving is 24.57% (*standard deviation 0.034*). Figure 60 shows a 100 runs of sample with size of 25%.

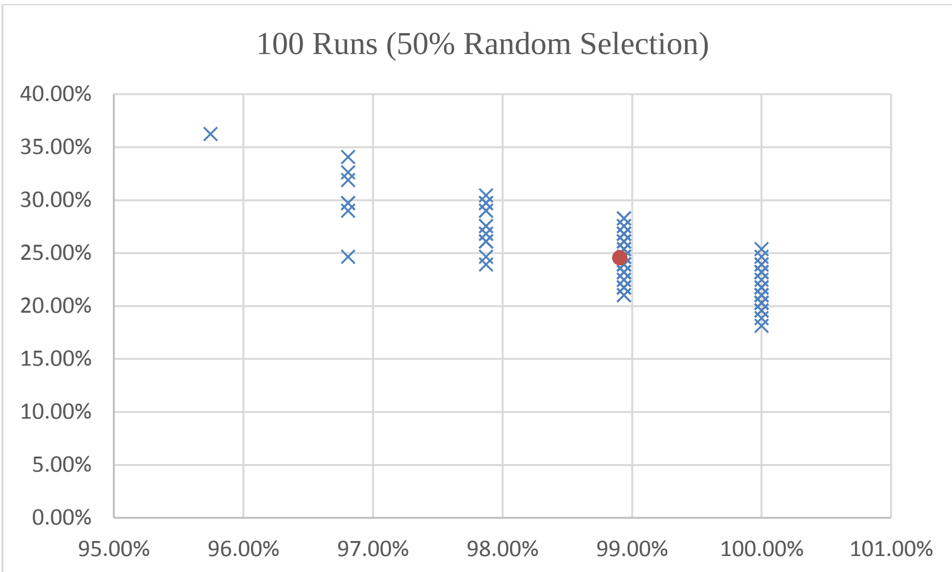


Figure 60: fattoriale specification random selection (50%)

7.5.8 Results Summary

Table 49 describes the summary of the results of the application of N-selective.

Table 49: 2, 4, 6-N-selective results for the case studies

	ferrymanSimulator	railroadGate	sluiceGateGround	cruiseControl	fattoriale

2 selective	Operators	ETR RRO	ROR ICR	RRO TRR	TRR RGCR	ARO RRO
	Effectiveness	100%	100%	100%	99.12%	98.94%
	Saving	30.14%	19.67%	55%	42.42%	40.58%
4 selective	Operators	ETR RRO CRRO ARO	ROR ICR ERR RRO	RRO TRR ARO RGCR	TRR RGCR RRO LOR	ARO RRO ROR ABS
	Effectiveness	98.26%	100%	99.24%	98.53%	96.81%
	Saving	56.81%	28.42%	75.5%	66.62%	52.17%
6 selective	Operators	ETR RRO CRRO ARO RTS TTR	ROR ICR ERR RRO TRR STF	RRO TRR ARO RGCR RTS STF	TRR RGCR RRO LOR ERR STF	ARO RRO ROR ABS ERR CTR
	Effectiveness	98.26%	100%	96.18%	98.53%	96.81%
	Saving	68.12%	38.25%	89%	80.09%	73.91%

Table 50 describes the summary of the results of the application of random selection.

Table 50: 10%, 25%, 50% random selection results for the case studies

		ferrymanSimulator	railroadGate	sluiceGateGround	cruiseControl	fattoriale
10% Random Selective	Effectiveness	96.77%	95.07%	96.42%	95.99%	93.69%
	Effectiveness Stability	0.032	0.049	0.02	0.021	0.034
	Saving	87.33%	85.55%	83.93%	87.30%	84.2%
	Saving Stability	0.013	0.018	0.015	0.011	0.027
25% Random Selective	Effectiveness	99.08%	99.62%	98.77%	98.47%	96.44%
	Effectiveness Stability	0.011	0.012	0.013	0.022	0.016
	Saving	68.82%	64.55%	60.13%	68.83%	62%
	Saving Stability	0.02	0.012	0.02	0.013	0.034
50%	Effectiveness	99.91%	100%	99.84%	99.48%	98.90%

Random Selective	Effectiveness Stability	0.004	0	0.015	0.004	0.01
	Saving	37.65%	29.9%	20.35%	38.13%	24.57%
	Saving Stability	0.018	0.023	0.02	0.016	0.034

7.6 Overall Operator-Based Selection Mutation

Typical Operator-based selection reduces the number of generated mutants, however, the mutants are actually generated but the output of the generation process to the testing process would be reduced based on which operators must be eliminated. In other words, the set of operators that would be discarded can be obtained without generating the full set of mutants. Thus, in order to reduce the computation cost further, we have investigated the possibility of generalizing the operator-based selection considering all of the case studies. In order to carry out our investigation, we must first determine the list of operators responsible for generating the largest share of mutants against all case studies. It should be noted that the weight of each operator would be considered as the ratio generated mutants per operator per case study rather than the total number of generated mutants for all case studies. Table 51 provides a top six operators ranked list (*RRO, CTR, TRR, ARO, RGCR, and ROR*) of weight per operators. These Operators produce on average 58% of the total generated mutants (*based on the selected case studies*).

Table 51: Ranking dominant operators (All case studies)

	Ferryman	railroadGate	sluiceGateGround	cruiseControl	AdvancedClock	AdvancedClock 2	fattoriale	Total
RRO	42	12	66	48	22	24	24	238
CTR	0	0	2	0	80	51	6	139
TRR	11	12	44	96	11	20	0	194
ARO	19	7	36	13	13	20	37	145
RGCR	5	8	12	72	2	6	6	111
ROR	8	19	2	8	10	9	15	71

	Ferryman	railroadGate	sluiceGateGround	cruiseControl	AdvancedClock	AdvancedClock 2	fattoriale	Average	
RRO	15%	9%	32%	11%	10%	11%	17%	15%	**57%**
CTR	0%	0%	1%	0%	37%	24%	4%	9%	
TRR	4%	9%	21%	23%	5%	10%	0%	10%	
ARO	7%	5%	17%	3%	6%	10%	26%	11%	
RGCR	2%	6%	6%	17%	1%	3%	4%	6%	
ROR	3%	14%	1%	2%	5%	4%	11%	6%	

Only operator-based selection will be investigated since the random selection is not applicable for generalization. We have performed 2-selective, 4-selective, and 6-selective for each study case as follows:

2-selective

As shown in Table 51, the two most dominant operators are RRO, and CTR. Table 52 shows the results of eliminating the mutants generated by these operators.

Table 52: Overall 2-Operators Selection mutation

	Ferryman	railroadGate	sluiceGateGround	cruiseControl	fattoriale
Effectiveness	100.00%	100.00%	100.00%	100.00%	100.00%
Saving	15.94%	4.37%	30.00%	11.05%	27.54%

4-selective

While the four most dominant operators are RRO, CTR, TRR, and ARO. Table 53 shows the results of eliminating the mutants generated by these operators.

Table 53: Overall 4-Operators Selection mutation

	Ferryman	railroadGate	sluiceGateGround	cruiseControl	fattoriale
Effectiveness	98.26%	100.00%	100.00%	100.00%	98.94%
Saving	24.35%	10.93%	65.00%	34.47%	49.28%

6-selective

Considering the six most dominant operators, which are RRO, CTR, TRR, ARO, RGCR, and ROR. Table 54 shows the results of eliminating the mutants generated by these operators.

Table 54: Overall 6-Operators Selection mutation

	Ferryman	railroadGate	sluiceGateGround	cruiseControl	fattoriale
Effectiveness	98.26%	100.00%	99.24%	99.12%	96.81%
Saving	28.41%	27.87%	77.50%	61.40%	61.59%

7.7 General Discussion

In this section, we have addressed the aforementioned questions as follows

Q1[56]: What are the most dominant mutation operators out of the proposed AsmetaL mutation operators?

Based on section 7.6, RRO, CTR, TRR, ARO, RGCR, and ROR are the most dominant operators that are responsible of 58% of the total number of generated mutants. Table 51 shows the amount and percentage of each dominant operator.

Q2[55][56]: Is N-selective mutation applicable in the context AsmetaL?

We compare the results of 2-, 4-, 6- selective obtained for each case study individually with the results obtained by other researches (for other languages). As shown in Table 49, the average 2-selective operator based effectiveness is 99.61%, while the average saving is 37.56%. Mathur [54] has obtained 99.99% effectiveness and 24% saving. It is noticeable that the effectiveness achieved in for 2-selective in the context of ASM is slightly less, however, the saving achieved is fairly higher. If RRO and CTR are considered for the 2-selective, as shown in Table 52, the obtained average level of effectiveness is 100%, while the average saving is 17.78%.

The 4- selective average of effectiveness is 98.57%, while the average saving is 55.9%. Comparing the obtained results with results obtained by Offutt [55] research (*99.84% effectiveness and 41% saving*), It is noticeable that the level of effectiveness achieved in for 4-selective in the context of ASM is slightly less, however, the saving achieved is fairly higher. If RRO, CTR, TRR, and ARO are considered for the 4-selective, as shown in Table 53, the obtained average level of effectiveness is 99.44%, while the average saving is 36.81%.

The 6- selective average of effectiveness is 97.96%, while the average saving is 69.87%. Comparing the obtained results with results obtained by Offutt [56] research (*88.71% effectiveness and 60% saving*), It is noticeable that the level of effectiveness and saving achieved in for 6-selective in the context of ASM is dramatically better. If RRO, CTR, TRR, ARO, RGCR, and ROR are considered for the 6-selective, as shown in Table 54, the obtained average level of effectiveness is 98.69%, while the average saving is 51.35%.

Based on the comparison above, we consider that N-selective is applicable in the context of ASM.

Q3: Is random-based selective mutation applicable in the context Abstract State Machines?

We based our answer on the results from section 7.5, as shown in Table 50, the average level of effectiveness obtained by 10% random selection is 95.59%, where the average stability factor for the effectiveness (*100 run standard deviation*) is 0.031, In addition, the average level of saving is 85.67%, where the average stability factor for the saving is 0.0168. Comparing our results with Wong and Mathur [51] research (10% selective, level of effectiveness is 84%), our results achieves dramatically better effectiveness score.

In case of 25% random selection, the average level of effectiveness is 98.48%, where the average stability of effectiveness is 0.015. In addition, the average level of saving is 64.87%, where the average stability of saving is 0.02.

In addition, in case of 50% random selection, the average level of effectiveness is 99.63%, where the average stability of effectiveness is 0.007. In addition, the average level of saving is 30.12%, where the average stability of saving is 0.022.

Based on our case study results, we can consider that random based selection is applicable in the context of ASM.

Q4[98]: How do operator-based and random-based mutant selection compare in the context of Abstract State Machines?

Ultimately, the relationship between effectiveness and savings is a tradeoff relationship. As described in the answer to **Q2**, the order of the 2, 4, and 6 N selective is descending order in term of effectiveness, however, it is ascending in term of saving. In contradiction, as described in the answer to **Q3**, the order of 10%, 25%, and 50% random selective is ascending in term of effectiveness, whereas in term of saving, it is descending order. Hence, we compare, in term of effectiveness and saving, 2- N selective with 50% random selective, 4- N selective with 25% random selective, and 6- N selective with 10% random selective.

In case of 2 – N selective and 50% random selection, random selective (99.63%) preform slightly better than 2 – N selective (99.61%) in term of effectiveness. However in term of saving, 2 – N selective (37.56%) preform fairly better than random selective (30.12%).

In case of 4 – N selective and 25% random selection, 4 – N selective (98.57%) preform slightly better than random selective (98.48%) in term of effectiveness. However in term of saving, random selective (64.87%) preform fairly better than 4 – N selective (55.9%).

In case of 6 – N selective and 10% random selection, 6 – N selective (97.96%) preform fairly better than random selective (95.59%) in term of effectiveness. However in term of saving, random selective (85.67%) preform dramatically better than 6 – N selective (64.87%).

As mentioned earlier, the relationship between effectiveness and savings is a tradeoff relationship. The selected case studies are insufficient to answer that question. It is worth noting that random-based selection provides more fixable ratio selection that can be subjective to the user need.

Q5[99]: Does random-based mutant selection provide a stable mutation scores in the context of AsmetaL?

The stability calculation is based on standard deviation which indicates how far the collected data from each other. It used as measurement of data precision. In the random selection mutation, it is noticeable that the stability measurement does not exceed 0.05 for both effectiveness and cost saving for the case studies as shown in Table 50. It is observed that 10% random selective analysis results in higher effectiveness standard deviation in respect with 25% and 50% random mutation. However, it results in lower cost saving standard deviation in respect with 25% and 50% random mutation. In addition, 50% selective mutation is in contrast. Thus, we can consider our results (*100 random sample runs*) stable.

7.8 Threats to Validity

In this section we have addressed any possible threats to validity in our thesis as follows:

Construct Validity: is concerned with the relevance and the meaningfulness of the used measures. In order to reduce threats of construction validity we have used metrics to measure the selective reduction techniques used by many other studies. Another threat to validity is the manual checking of equivalent mutants, which is a tedious and error prone activity. Many studies, (*e.g., [55], [56], [98], [99], and [103]*), treated the remaining alive mutants after refining test suites as equivalent mutants, and thereby they are discarded.

Internal Validity: is concerned with the uncontrolled variables used in experiments. In order to reduce internal threats to validity, we have implemented MuAsmetaL to enforce the consistency of data collection. All the results of the case studies were collected using MuAsmetaL, thus, eliminating any faults related to manual data collection. Second, threat to internal validity is related to the use of MuAsmetaL. The tool is still in the prototype stage and requires more testing and improvements. To reduce this risk, selected test cases are executed using the tool and manually, showing no discrepancies. Third, MuAsmetaL does not have the ability to detect equivalent mutants nor it consider them in the mutation score calculation.

External Validity: is concerned with how well you can generalize from the results of one study to the real world. The ability to generalize depends on how similar the study environment is to that use in actual practice. In order to reduce external threats to validity, we have chosen several case studies obtained from the literature. Case studies that represent a diversity of AsmetaL specifications in term of specification size and level of abstraction. However, not all operators produce mutants due to the absence of certain AsmetaL constructs. All operators were implemented in MuAsmetaL and can be used other case studies. In addition, we have excluded non-deterministic specification from our selection, since; testing non-deterministic behavior is off scope.

Last threat to external validity of the results reported in the case studies may be related to the fact that ATGT does not fully support the AsmetaL language. However, our approach does not depend on the exclusive use of the ATGT tool. Test cases can be generated using any tool or even created manually.

CHAPTER 8

Conclusions and Future Work

The aim of this thesis is to propose a mutation-testing approach for Abstract State Machines paradigm. The work described in this thesis has been concerned with the design and evaluation of mutation operators for AsmetaL language, which is considered as incarnation of ASMs concept. A set of 49 mutation operators, (each is associated with an AsmetaL potential fault), are classified into 5 categories, have been proposed. An empirical investigation, that demonstrates the applicability of mutation testing in the context of ASMs, is presented in chapter 6. In addition, the effectiveness of operator-based and random-based selection, in order to reduce the computational cost of mutation testing, are investigated in chapter 7.

8.1 Hypothesis of the Thesis

To conclude our research, the research hypotheses are recalled

<u>Research Hypothesis 1:</u>

Our first research hypothesis is denoted as follows:

> "Mutation testing can be applied to the Abstract State Machines (ASM) formalism. This can be achieved through the design and the application of ASM-based mutation operators."

Based on our approach and empirical evaluation, it can be noticed that the proposed mutation operators for AsmetaL are able to generate a set of syntactically valid mutants. Thesis mutants mimic potential fault that may exist. We can observe, based on case studies, that most of the generated mutant are killable. In this sense, the application of mutation testing achieves its goals, thus, we can conclude that mutation testing is applicable in the context of Abstract State Machines.

Research Hypothesis 2:

Our second research hypothesis is denoted as follows:

> "ASM-based mutation testing is an effective approach to assess the adequacy of ASM-based test suites."

We can observe, based on case studies, that the ability of test cases to kill mutants vary from one to another, hence, we can judge the effectiveness of test cases based on the proposed operators, furthermore, we can compare the effectiveness of two test cases. Our drawn conclusion is mutation testing is an effective approach to assess the adequacy of ASM-based test suites.

Research Hypothesis 3:

Our Third research hypothesis is denoted as follows:

> "Mutation-based testing cost reduction techniques, such as selective and random mutation can be applied in the context of Abstract State Machines specifications."

We have performed selective and random mutation techniques, in chapter 7. Our judgment would be based on levels of effectiveness and savings for several case studies. Despite the fact that ASM context is different from other programming language such as C and Java, we compare our results with other studies. Our obtained results are to other works. Therefore, the drawn conclusion is that selective and random mutation are applicable in the context of Abstract State Machines specifications.

8.2 Thesis Contributions of the Thesis

To conclude our research, the thesis contributions are recalled

8.2.1 Contribution 1: Design and Evaluation of Mutation Operators for the AsmetaL language

We have proposed a set of 49 operators for the AsmetaL language. The resulting operators are categorized into 5 categories targeting different types of AsmetaL faults. Each mutation operator is described using a concrete example and analyzed with respect to the produced mutants (*e.g., valid/invalid, equivalent/non-equivalent, etc*.). Furthermore, a mathematical characterization of the upper bound of the number of generated mutants is provided for each operator. Chapter 4 presented and discusses the set of proposed AsmetaL-based mutation operators.

8.2.2 Contribution 2: Empirical Evaluation of the Proposed Approach

Our proposed mutation-based approach is evaluated empirically using a set of 7 case studies of different sizes. We have shown that mutation testing can be applied effectively to ASM-based specifications. Furthermore, as an application of the proposed approach and since the only tool, spotted in the literature, that supports the generation of test cases for AsmetaL language is ATGT, we have focused on the evaluation of the test suites produced using the ATGT coverage criteria. We have shown that some ATGT coverage criteria are more adequate than others are. Chapter 6 presents and discusses our empirical experiments.

8.2.3 Contribution 3: Development of MuAsmetaL

We have developed a prototype tool (*called MuAsmetaL*) to perform AsmetaL-based mutation testing. The tool presents many features that can be summarized as follows:

- Generating mutants based on the proposed operators.
- Validating the correctness of all the generated mutants using AsmetaLc.
- Validating syntactic equivalency of generated mutants against the original specification.
- Running test cases against the original specification.
- Running test cases against mutants.
- Calculating mutation score per operator and for all mutants.

Chapter 5 presents our MuAsmetaL tool.

8.2.4 Contribution 4: Investigation of Cost Reduction Techniques in the ASM Context

Mutation testing is known to have a high computation cost due to the large number of generated mutants. Many techniques have been proposed to reduce the cost of the application of mutation testing. In this thesis, we have applied random mutation and selective mutation to AsmetaL specifications. As discussed in Chapter 7 , we were able to achieve satisfactory results with respect to the resulting mutation score and the cost savings.

8.3 Future Work

In this section, we present some of the works that can be done as complementary to the proposed AsmetaL mutation approach:

- **The conduction of wider empirical studies**

Experiments with varieties of subjects can be conducted in order to provide conclusion that can be generalized. In addition, thresholds can be drawn based on the outputs of that empirical studies, since, ASM has its own unique context.

- **Test case generation techniques adequacy assessment**

  The proposed approach can be used to assess the adequacy of any AsmetaL test case generation techniques. In addition, the usage of AsmetaL promotes the comparison between them. In this thesis, we have compared different ATGT test coverage criteria, it can be used with other test generation tools.

- **Intermediate state 'weak' mutation testing**

  Introducing compiler-based reduction technique based on intermediate state 'weak' mutation testing. It can reduce computation cost by forcing the machine to the desired state (*precondition state*).

- **AsmetaL test cases prioritization**

  The proposed mutation based testing for AsmetaL can be used to prioritize test cases based on internal metrics and generation criteria in order to determine the execution sequence. This prioritized sequence will reduce computation cost and provides an optimized test process.

- **Test case generation/Equivalency analysis**

Test case generation/Equivalency analysis using AsmetaL mutation testing and model checker counter example. Generally, equivalency analysis is undecidable problem. Many proposed approach (*e.g., Laser equivalent mutation detection [118]*) combines mutation testing and model checker to provide a fully automated tool that can generated mutants, detect equivalent mutants, use model checker counter-example to generate new test case. Therefore, mutation testing not only assess the adequacy of test suites, but improve it.

References

[1] Adrion, W. Richards, Martha A. Branstad, and John C. Cherniavsky. "Validation, verification, and testing of computer software." *ACM Computing Surveys (CSUR)* 14.2 (1982): 159-192.

[2] Börger, Egon, and Robert F. Stärk. *Abstract State Machines: A Method for High-level System Design and Analysis; with 19 Tables*. Springer, 2003.

[3] Gurevich, Yuri. "Reconsidering Turing's thesis:(toward more realistic semantics of programs)." (1984).

[4] Börger, Egon. "The origins and the development of the ASM method for high level system design and analysis." *Journal of Universal Computer Science* 8.1 (2002): 2-74.

[5] Richardson, Debra J., Stephanie Leif Aha, and T. Owen O'malley. "Specification-based test oracles for reactive systems." *Proceedings of the 14th international conference on Software engineering*. ACM, 1992.

[6] Farahbod, Roozbeh, and Uwe Glässer. "The CoreASM modeling framework." *Software: Practice and Experience* 41.2 (2011): 167-178.

[7] Hassine, Jameleddine. *Formal semantics and verification of use case maps*. Diss. Concordia University, 2008.

[8] Gurevich, Yuri. "Sequential abstract-state machines capture sequential algorithms." *ACM Transactions on Computational Logic (TOCL)* 1.1 (2000): 77-111.

[9] Blass, Andreas, and Yuri Gurevich. "Abstract state machines capture parallel algorithms." *ACM Transactions on Computational Logic (TOCL)* 4.4 (2003): 578-651.

[10] http://research.microsoft.com/en-us/projects/specexplorer/

[11] Barnett, Mike, et al. "Towards a tool environment for model-based testing with AsmL." *Formal Approaches to Software Testing*. Springer Berlin Heidelberg, 2004. 252-266.

[12] http://www.coreasm.org/download.php

[13] Farahbod, Roozbeh, and Uwe Glässer. "The CoreASM modeling framework." *Software: Practice and Experience* 41.2 (2011): 167-178.

[14] http://asmeta.sourceforge.net/

[15] Gargantini, Angelo, Elvinia Riccobene, and Patrizia Scandurra. "Ten reasons to metamodel ASMs." *Rigorous Methods for Software Construction and Analysis*. Springer Berlin Heidelberg, 2009. 33-49.

[16] Gargantini, Angelo, Elvinia Riccobene, and Patrizia Scandurra. "A Metamodel-based Language and a Simulation Engine for Abstract State Machines." *J. UCS* 14.12 (2008): 1949-1983.

[17] Gargantini, Angelo, Elvinia Riccobene, and Patrizia Scandurra. "Metamodelling a formal method: applying mde to abstract state machines." (2006).

[18] http://www2.cs.uni-paderborn.de/cs/asm/ASMToolPage/asm-workbench.html

[19] Del Castillo, Giuseppe. *The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models*. Springer Berlin Heidelberg, 2001.

[20] Gargantini, Angelo, Elvinia Riccobene, and Patrizia Scandurra. "A metamodel-based simulator for ASMs." *Proc. of the 14th Intl. Abstract State Machines Workshop*. 2007.

[21] http://fmse.di.unimi.it/asmeta/download/AsmetaL_quickguide.html

[22] http://fmse.di.unimi.it/asmeta/download/AsmetaL_guide.pdf

[23] http://fmse.di.unimi.it/asmeta/download/AsmetaL_EBNF.html

[24] http://asmeta.sourceforge.net/download/asmetalc.html

[25] http://asmeta.sourceforge.net/download/asmetas.html

[26] http://asmeta.sourceforge.net/download/asmetav.html

[27] Arcaini, Paolo, Angelo Gargantini, and Elvinia Riccobene. "AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications." *Abstract State Machines, Alloy, B and Z*. Springer Berlin Heidelberg, 2010. 61-74.

[28] Arcaini, Paolo, Angelo Gargantini, and Elvinia Riccobene. "A model advisor for NuSMV specifications." *Innovations in systems and software engineering* 7.2 (2011): 97-107.

[29] http://asmeta.sourceforge.net/download/asmetasmv.html

[30] Gargantini, Angelo, Elvinia Riccobene, and Patrizia Scandurra. "AsmEE: an Eclipse plug-in in a metamodel based framework for the Abstract State Machines." *International Conference on Eclipse Technologies (Eclipse IT). Napoli (Italy)*. 2007.

[31] http://fmse.di.unimi.it/asmee/update/

[32] Scandurra, Patrizia, et al. "Functional requirements validation by transforming use case models into Abstract State Machines." *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012.

[33] http://nusmv.fbk.eu/

[34] http://code.google.com/a/eclipselabs.org/p/nuseen/

[35] http://code.google.com/a/eclipselabs.org/p/nusmv-tools/

[36] http://cs.unibg.it/gargantini/software/atgt/

[37] Arcaini, Paolo, Angelo Gargantini, and Elvinia Riccobene. "Optimizing the automatic test generation by SAT and SMT solving for Boolean expressions." *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011.

[38] http://fmse.di.unimi.it/atgtBoolean.html

[39] Albani, Fabio, Elvinia Riccobene, and Patrizia Scandurra. "An Eclipse-based SCA design framework to support coordinated execution of services." *Proc. of the 6th Workshop of the Italian Eclipse Community*. 2011.

[40] Jeevarathinam, R., and Antony Selvadoss Thanamani. "A survey on mutation testing methods, fault classifications and automatic test cases generation." *Journal of Scientific & Industrial Research* 70 (2011): 113-117.

[41] Lipton, R. "Fault diagnosis of computer programs." *Student Report, Carnegie Mellon University* (1971).

[42] Acree, Allen T., et al. *Mutation Analysis*. No. GIT-ICS-79/08. GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1979.

[43] DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward. "Hints on test data selection: Help for the practicing programmer." *Computer* 11.4 (1978): 34-41.

[44] Jia, Yue, and Mark Harman. "An analysis and survey of the development of mutation testing." *Software Engineering, IEEE Transactions on* 37.5 (2011): 649-678.

[45] Baldwin, Douglas, and Frederick Sayward. *Heuristics for Determining Equivalence of Program Mutations*. GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1979.

[46] Offutt, A. Jefferson, and Jie Pan. "Detecting equivalent mutants and the feasible path problem." *Computer Assurance, 1996. COMPASS'96,'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*. IEEE, 1996.

[47] Harman, Mark, Rob Hierons, and Sebastian Danicic. "The relationship between program dependence and mutation analysis." *Mutation testing for the new century*. Springer US, 2001. 5-13.

[48] Ellims, Michael, Darrel Ince, and Marian Petre. "The csaw c mutation tool: Initial results." *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007.

[49] Grun, Bernhard JM, David Schuler, and Andreas Zeller. "The impact of equivalent mutants." *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. IEEE, 2009.

[50] Acree Jr, Allen Troy. *On Mutation*. No. GIT-ICS-80/12. GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1980.

[51] Mathur, Aditya P., and W. Eric Wong. "An empirical comparison of data flow and mutation-based test adequacy criteria." *Software Testing, Verification and Reliability* 4.1 (1994): 9-31.

[52] Hussain, Shamaila. "Mutation clustering." *Ms. Th., King's College London, Strand, London* (2008).

[53] Faber, Vance. "Clustering and the continuous k-means algorithm." *Los Alamos Science* 22 (1994): 138-144.

[54] Mathur, Aditya P. "Performance, effectiveness, and reliability issues in software testing." *Computer Software and Applications Conference, 1991. COMPSAC'91., Proceedings of the Fifteenth Annual International*. IEEE, 1991.

[55] Offutt, A. Jefferson, Gregg Rothermel, and Christian Zapf. "An experimental evaluation of selective mutation." *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, 1993.

[56] Offutt, A. Jefferson, et al. "An experimental determination of sufficient mutant operators." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5.2 (1996): 99-118.

[57] Jia, Yue, and Mark Harman. "Constructing subtle faults using higher order mutation testing." *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*. IEEE, 2008.

[58] Polo, Macario, Mario Piattini, and Ignacio García-Rodríguez. "Decreasing the cost of mutation testing with second-order mutants." *Software Testing, Verification and Reliability* 19.2 (2009): 111-131.

[59] DeMillo, Richard A., Edward W. Krauser, and Aditya P. Mathur. "Compiler-integrated program mutation." *Computer Software and Applications Conference, 1991. COMPSAC'91., Proceedings of the Fifteenth Annual International*. IEEE, 1991.

[60] Woodward, M. R., and K. Halewood. "From weak to strong, dead or alive? an analysis of some mutation testing issues." *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*. IEEE, 1988.

[61] King, Kim N., and A. Jefferson Offutt. "A fortran language system for mutation-based software testing." *Software: Practice and Experience* 21.7 (1991): 685-718.

[62] Delamare, Romain, Benoit Baudry, and Yves Le Traon. "AjMutator: A tool for the mutation analysis of AspectJ pointcut descriptors." *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. IEEE, 2009.

[63] Tuya, Javier, Ma José Suárez-Cabal, and Claudio de la Riva. "Mutating database queries." *Information and Software Technology* 49.4 (2007): 398-417.

[64] Ma, Yu-Seung, Jeff Offutt, and Yong Rae Kwon. "MuJava: an automated class mutation system." *Software Testing, Verification and Reliability* 15.2 (2005): 97-133.

[65] Bogacki, Bartosz, and Bartosz Walter. "Evaluation of test code quality with aspect-oriented mutations." *Extreme Programming and Agile Processes in Software Engineering*. Springer Berlin Heidelberg, 2006. 202-204.

[66] Belinfante, Axel, Lars Frantzen, and Christian Schallhart. "14 Tools for Test Case Generation." *Model-Based Testing of Reactive Systems*. Springer Berlin Heidelberg, 2005. 391-438.

[67] Lee, David, and Mihalis Yannakakis. "Principles and methods of testing finite state machines-a survey." *Proceedings of the IEEE* 84.8 (1996): 1090-1123.

[68] Barnett, Mike, et al. "Towards a tool environment for model-based testing with AsmL." *Formal Approaches to Software Testing*. Springer Berlin Heidelberg, 2004. 252-266.

[69] Stärk, Robert F., Dipl-Inf Joachim Schmid, and Egon Börger. "Abstract State Machines." *Java and the Java Virtual Machine*. Springer Berlin Heidelberg, 2001. 15-28.

[70] Grieskamp, Wolfgang, et al. "Conformance testing with abstract state machines." (2001).

[71] Bochmann, Gregor V., and Alexandre Petrenko. "Protocol testing: review of methods and relevance for software testing." *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 1994.

[72] Gonenc, Guney. "A method for the design of fault detection experiments." *Computers, IEEE Transactions on* 100.6 (1970): 551-558.

[73] Chow, Tsun S. "Testing software design modeled by finite-state machines." *Software Engineering, IEEE Transactions on* 3 (1978): 178-187.

[74] Sabnani, Krishan, and Anton Dahbura. "A protocol test generation procedure." *Computer Networks and ISDN systems* 15.4 (1988): 285-297.

[75] Gargantini, Angelo, and Elvinia Riccobene. "ASM-based testing: Coverage criteria and automatic test sequence generation." *Journal of Universal Computer Science* 7.11 (2001): 1050-1067.

[76] Clarke, Edmund M., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT press, 1999.

[77] Gargantini, Angelo, Elvinia Riccobene, and Salvatore Rinzivillo. "Using Spin to generate tests from ASM specifications." *Abstract State Machines 2003*. Springer Berlin Heidelberg, 2003.

[78] Grieskamp, Wolfgang, et al. "Test case generation from AsmL specifications." *Abstract State Machines 2003*. Springer Berlin Heidelberg, 2003.

[79] Pinto Ferraz Fabbri, S. C., et al. "Mutation analysis testing for finite state machines." *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*. IEEE, 1994.

[80] Fabbri, Sandra Camargo Pinto Ferraz, et al. "Mutation testing applied to validate specifications based on statecharts." *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*. IEEE, 1999.

[81] Batth, Samrat S., et al. "Specification of timed EFSM fault models in SDL." *Formal Techniques for Networked and Distributed Systems–FORTE 2007*. Springer Berlin Heidelberg, 2007. 50-65.

[82] Hassine, Jameleddine. "Abstract State Machines Mutation Operators." *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*. 2012.

[83] Hassine, Jameleddine. "Design and Classification of Mutation Operators for Abstract State Machines." *International Journal On Advances in Software* 6.1 and 2 (2013): 80-91.

[84] Hierons, Robert M., and Mercedes G. Merayo. "Mutation testing from probabilistic and stochastic finite state machines." *Journal of Systems and Software* 82.11 (2009): 1804-1818.

[85] Li, Jin-hua, Geng-xin Dai, and Huan-huan Li. "Mutation analysis for testing finite state machines." *Electronic Commerce and Security, 2009. ISECS'09. Second International Symposium on*. Vol. 1. IEEE, 2009.

[86] Fabbri, Sandra Camargo Pinto Ferraz, et al. "Mutation testing applied to validate specifications based on petri nets." *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*. Chapman & Hall, Ltd., 1995.

[87] De Souza, Simone Do Rocio Senger, et al. "Mutation testing applied to estelle specifications." *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*. IEEE, 2000.

[88] Ammann, Paul, and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[89] Gargantini, A., Calvagna, A., Riccobene, E., Rinzivillo, S., Galati, S.: ATGT: ASM Tests Generation Tool. http://cs.unibg.it/gargantini/software/atgt/ (2013)

[90] Carioni, Alessandro, et al. "A scenario-based validation language for ASMs." *Abstract State Machines, B and Z*. Springer Berlin Heidelberg, 2008. 71-84.

[91] Holzmann, Gerard J. "The model checker SPIN." *IEEE Transactions on software engineering* 23.5 (1997): 279-295.

[92] Gargantini, Angelo. "Using model checking to generate fault detecting tests." *Tests and Proofs*. Springer Berlin Heidelberg, 2007. 189-206.

[93] Calvagna, Andrea, and Angelo Gargantini. "A logic-based approach to combinatorial testing with constraints." *Tests and proofs*. Springer Berlin Heidelberg, 2008. 66-83.

[94] Calvagna, Andrea, and Angelo Gargantini. "Combining satisfiability solving and heuristics to constrained combinatorial interaction testing." *Tests and Proofs*. Springer Berlin Heidelberg, 2009. 27-42.

[95] Woodward, Martin R. "Errors in algebraic specifications and an experimental mutation testing tool." *Software Engineering Journal* 8.4 (1993): 211-224.

[96] Rad, Mohsen Falah, and Mohadeseh Moosavi. "Investigation of Improving Test Data in Mutation Testing by Optimization Methods."

[97] Baier, Christel, and Joost-Pieter Katoen. *Principles of model checking*. Vol. 26202649. Cambridge: MIT press, 2008.

[98] Gligoric, Milos, et al. "Selective mutation testing for concurrent code." *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013.

[99] Zhang, Lu, et al. "Is operator-based mutant selection superior to random mutant selection?." *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010.

[100] Zhang, Lingming, et al. "Operator-based and random mutant selection: Better together." *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013.

[101] Mresa, Elfurjani S., and Leonardo Bottaci. "Efficiency of mutation operators and selective mutation strategies: An empirical study." *Software Testing Verification and Reliability* 9.4 (1999): 205-232.

[102] Zhang, Lingming, Darko Marinov, and Sarfraz Khurshid. "Faster mutation testing inspired by test prioritization and reduction." *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013.

[103] Siami Namin, Akbar, James H. Andrews, and Duncan J. Murdoch. "Sufficient mutation operators for measuring test effectiveness." *Proceedings of the 30th international conference on Software engineering*. ACM, 2008.

[104] Namin, A. Siami, and James H. Andrews. "Finding sufficient mutation operators via variable reduction." *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*. 2006.

[105] Barbosa, Ellen Francine, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. "Toward the determination of sufficient mutant operators for C†." *Software Testing, Verification and Reliability* 11.2 (2001): 113-136.

[106] http://www.ccse.kfupm.edu.sa/~jhassine/MuAsmetaL/

[107] Copeland, Tom. *Generating parsers with JavaCC*. Alexandria: Centennial Books, 2007.

[108] Gargantini, A., Riccobene, E., Scandurra, P.: ferrymanSimulator AsmetaL specification. http://cs.unibg.it/gargantini/software/atgt/examples.zip (2013)

[109] Gargantini, A., Riccobene, E., Scandurra, P.: railroadGate AsmetaL specification. http://cs.unibg.it/gargantini/software/atgt/examples.zip (2013)

[110] Jonsson, Manfred Broy Bengt, Joost-Pieter Katoen Martin Leucker, and Alexander Pretschner. "Model-Based Testing of Reactive Systems." (2005).

[111] Gargantini, A., Riccobene, E., Scandurra, P.: sluiceGateGround AsmetaL specification. http://cs.unibg.it/gargantini/software/atgt/examples.zip (2013)

[112] Börger, Egon. "The abstract state machines method for high-level system design and analysis." *Formal Methods: State of the Art and New Directions*. Springer London, 2010. 79-116.

[113] Gargantini, A., Riccobene, E., Scandurra, P.: cruiseControl AsmetaL specification. http://cs.unibg.it/gargantini/software/atgt/examples.zip (2013)

[114] Kirby, James. *Example NRL/SCR software requirements for an automobile cruise control and monitoring system*. Technical report, Wang Institute of Graduate Studies, 1988.

[115] Gargantini, A., Riccobene, E., Scandurra, P.: AdvancedClock AsmetaL specification. http://cs.unibg.it/gargantini/software/atgt/examples.zip (2013)

[116] Gargantini, A., Riccobene, E., Scandurra, P.: AdvancedClock2 AsmetaL specification. http://cs.unibg.it/gargantini/software/atgt/examples.zip (2013)

[117] Gargantini, A., Riccobene, E., Scandurra, P.: fattoriale AsmetaL specification. http://cs.unibg.it/gargantini/software/atgt/examples.zip (2013)

[118] du Bousquet, Lydie, and Michel Delaunay. "Towards mutation analysis for Lustre programs." *Electronic Notes in Theoretical Computer Science* 203.4 (2008): 35-48.

Vitae

Name :Osama Jamil AlKrarha

Nationality :Jordanian

Date of Birth :11/29/1986

Email :alkrarha@kfupm.edu.sa

Address :Al Khobar, Eastern Region, Saudi Arabia

Academic Background :Bachelor of Science in Software Engineering, King Fahd University of Petroleum and Minerals, currently works as research assistant in research institute at King Fahd University of Petroleum and Minerals.