STRONG MUTATION-BASED TEST GENERATION OF XACML POLICIES

by

Roshan Shrestha

A thesis
submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Boise State University

December 2018
DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Roshan Shresthat

Thesis Title: Strong Mutation-Based Test Generation of XACML Policies

Date of Final Oral Examination: 24 October 2018

The following individuals read and discussed the thesis submitted by student Roshan Shrestha, and they evaluated the student’s presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Dianxiang Xu, Ph.D. Chair, Supervisory Committee

Edoardo Serra, Ph.D. Member, Supervisory Committee

Yantian Hou, Ph.D. Member, Supervisory Committee

The final reading approval of the thesis was granted by Dianxiang Xu, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.
ACKNOWLEDGMENTS

I would like to thank Boise State University for giving me the opportunity to pursue graduate study without which I would not have come this far. I would like to express the highest gratitude to the Computer Science Department of Boise State University for always supporting me to achieve academic excellence.

I am grateful and have great appreciation to Dr. Dianxiang Xu for including me in his research group. I highly admire the learning opportunity and environment he provided to me. His ideas and suggestions were crucial for me to achieve my goals. His achievements, his dedication and his passion for the research in Computer Security will always be a continuous source of inspiration for the rest of my life.

I am also very grateful to Dr. Edoardo Serra and Dr. Yantian Hou for being on my thesis committee and providing valuable feedback on my work.

Finally, I am grateful to my parents and my sister for always supporting me and encouraging me to achieve my goals.
ABSTRACT

There exist various testing methods for XACML policies which vary in their overall fault detection ability and none of them can detect all the (killable) injected faults except for the simple policies. Further, it is unclear that what is essential for the fault detection of XACML policies. To address these issues, we formalized the fault detection conditions in the well-studied fault model of XACML policies so that it becomes clear what is essential for the fault detection. We formalized fault detection conditions in the form of reachability, necessity and propagation constraint. We, then, exploit these constraints to generate a mutation-based test suite with the goal to achieve perfect mutation score. Additionally, we have empirically evaluated the cost-effectiveness of various coverage-based testing methods against the near-optimal test suite from strong mutation-based test generation (SMT). Rule coverage has good cost-effectiveness such that it achieved better MKPT scores than SMT in many of the policies; however, it has poor fault detection capability. Decision coverage is nearly as cost-effective as SMT in most of the policies and it achieves better mutation score than rule coverage but could not achieve good mutation score in many of the policies. MC/DC have slightly less MKPT scores than SMT; nonetheless, among coverage-based testing methods, MC/DC tests have the highest mutation score and hence could reveal most of the faults. MC/DC even achieved a perfect mutation score for some policies; however, it still could not maintain good mutation score in all the policies.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ........................................................................................................ iv

ABSTRACT ............................................................................................................................ v

TABLE OF CONTENTS ......................................................................................................... vi

LIST OF TABLES ................................................................................................................... ix

LIST OF FIGURES ................................................................................................................ x

LIST OF ABBREVIATIONS .................................................................................................. xi

CHAPTER 1 ............................................................................................................................ 1

Introduction ......................................................................................................................... 1

1.1 Background ................................................................................................................... 1

1.2 Thesis Statement .......................................................................................................... 7

1.3 Method ......................................................................................................................... 9

1.4 Outline ......................................................................................................................... 14

CHAPTER 2 ............................................................................................................................ 15

Background and Related Work .......................................................................................... 15

2.1 Mutation analysis and mutation-based test generation ............................................. 15

2.2 Introduction to XACML ............................................................................................... 17

CHAPTER 3 ............................................................................................................................ 31

Fault Detection Condition ............................................................................................... 31

3.1 FDC for Change Rule Effect (CRE) ........................................................................... 35
3.2 FDC for Rule Target True (RTT).............................................................48
3.3 FDC for Rule Target False (RTF).........................................................59
3.4 FDC for Rule Condition True (RCT).....................................................65
3.5 FDC for Rule Condition False (RCF)....................................................72
3.6 FDC for Add Not Function (ANF)..........................................................78
3.7 FDC for Remove Not Function (RNF)....................................................82
3.8 FDC for Remove a Rule (RER)...............................................................83
3.9 FDC for Policy Target True (PTT)..........................................................89
3.10 FDC for Policy Target False (PTF).......................................................91
3.11 FDC for First Permit Rule (FPR)..........................................................93
3.12 FDC for First Deny Rule (FDR)............................................................94
3.13 FDC for Remove Parallel Target Element (RPTE)...............................94
3.14 FDC for Change Rule Combining Algorithm (CRC)............................100

CHAPTER 4 ....................................................................................................109

Mutation-Based Test Generation ..................................................................109

4.1 Strong Mutation-Based Test Generation.............................................110
4.2 Return Constraint .................................................................................117
4.3 FDC Constraint for Propagation .........................................................117
4.4 FDC Constraint and Tests Generation for RTT .................................121
4.5 FDC constraint and tests generation for RTF.....................................122
4.6 FDC Constraint and Test Generation for RCT....................................123
4.7 FDC Constraint and Test Generation for RCF....................................124
4.8 FDC Constraint and Test Generation for ANF .................................124
4.9 FDC Constraint and Test Generation for RNF ........................................124
4.10 FDC Constraint and Test Generation for RER ....................................124
4.11 FDC Constraint and Test Generation for PTT ....................................125
4.12 FDC Constraint and Test Generation for PTF ....................................125
4.13 FDC Constraint and Test Generation for FPR ....................................125
4.14 FDC constraint and test generation for FDR .....................................126
4.15 FDC Constraint and Test Generation for RPTE ................................127
4.16 FDC constraint and test generation for CRC ...................................129

CHAPTER 5 .................................................................................................130
Quantitative Analysis ...............................................................................130
5.1 Experiment Setup ...............................................................................130
5.2 Results ...............................................................................................133
5.3 Threats to Validity .............................................................................150

CHAPTER 6 .................................................................................................152
Conclusions ..............................................................................................152
6.1 Summary ............................................................................................152
6.2 Future work .......................................................................................153
REFERENCES ...........................................................................................155
LIST OF TABLES

Table 3.1: Fault Model ................................................................. 31
Table 3.2: A Faulty Policy with an Incorrect Rule Effect ......................... 36
Table 3.3: Possible Evaluations Of Other Rules when $R_{bi}$ Evaluates to True .... 39
Table 3.4: Possible Evaluations of Other Rules when $R_{bi}$ Evaluates to Error .......... 41
Table 3.5: Possible Evaluations of Other Rules When $R_{ti}$ Evaluates to $N/A$ and $R_i$ is 
Deny Rule in P ................................................................................. 53
Table 3.6. Possible Evaluations of Other Rules When $R_{ti}$ Evaluates to $N/A$ And $R_i$ is Permit Rule in P................... 54
Table 3.7: Possible Evaluations of Other Rules When $R_{ti}$ Evaluates to Error And $R_i$ is Deny Rule in P .......................................................... 55
Table 3.8. Possible Evaluations of Other Rules when $R_{ti}$ Evaluates to Error and $R_i$ is Permit Rule in P............................. 56
Table 5.1: Policies used for the experiment ............................................. 132
Table 5.2: Number of tests generated ..................................................... 135
Table 5.3: Test generation time (in milliseconds) ....................................... 138
Table 5.4: Mutation Scores ................................................................. 141
Table 5.5: Live Mutants ..................................................................... 144
Table 5.6: MKPT Scores ..................................................................... 148
LIST OF FIGURES

| Figure 1.1:   | ABAC Concept [1] | 3 |
| Figure 1.2:   | Data-flow in ABAC [2] | 5 |
| Figure 2.1:   | XACML policy language model [2] | 18 |
| Figure 2.2:   | A sample XACML policy [13] | 22 |
LIST OF ABBREVIATIONS

**XACML** - eXtensible Access Control Markup Language

**AC** - Access Control

**ABAC** - Attribute-based access control

**RBAC** - Role-based access control

**RCA** - Rule Combining Algorithm

**PCA** - Policy Combining Algorithm

**MKPT** - Mutants killed per unit test

**PT** - Policy Target

**PST** - Policy set Target

**PO** - Permit-overrides

**DO** - Deny-overrides

**PUD** - *permit-unless-deny*

**DUP** - *deny-unless-permit*

**FA** - *first-applicable*

**OPO** - Ordered *permit-overrides*

**ODO** - Ordered *deny-overrides*

**RC** - Rule Coverage

**NE-DC** - Non-error Decision Coverage

**DC** - Decision Coverage

**MC/DC** - Modified Condition/Decision Coverage

**NE-MC/DC** - Nor-*error* Modified Condition/Decision Coverage
FDC - Fault detection condition
PEP - Policy Enforcement Point
PDP - Policy Decision Point
PAP - Policy Administratio Point
PIP - Policy Information Point
I(D) - Indeterminate Deny
I(P) - Indeterminate Permit
I(DP) - Indeterminate Deny/Permit
SMT - Strong mutation test generation
NO-SMT - Non-optimized Strong mutation test generation
CRE - Change Rule Effect
RTT - Rule Target True
RTF - Rule Target False
RCT - Rule Condition True
RCF - Rule Condition False
ANF - Add Not Function
RNF - Remove Not Function
RER - Remove a rule
PTT - Policy Target True
PTF - Policy Target False
FPR - First Permit Rule
FDR - First Deny Rule
RPTE - Remove Parallel Target Element
CRC - Change Rule Combining Algorithm
CHAPTER 1

Introduction

1.1 Background

Access Control (AC) mechanism is a fundamental security mechanism that serves to limit the access to a system with virtual or physical resources (objects) based on subject requesting the access. In AC models like IBAC (Identity Based Access Control) or RBAC (Role-Based Access Control), the solution is based primarily on the identity of a subject where access to an object will be individually granted to a locally identified subject or locally defined roles that the subject is a member of. The subject qualifiers, such as identity and roles, are often insufficient in the expression of real-world AC needs because real-world AC requirements often need to deal with environmental conditions. However, traditional AC models like RBAC cannot incorporate factors pertaining to environmental conditions effectively. Environmental conditions include operational or situational context which are detectable environmental characteristics in which an access request is made. Environmental characteristics may include the current time, day of the week, a location of a user, or the current threat level which are independent of subject or object [1].

Attribute-Based Access Control (ABAC) has emerged as a new generation of access control methods for tackling the afore-mentioned issues. ABAC avoids the need for capabilities (operation/object pairs) to be directly assigned to a subject (requesters) or to their roles or groups before the request is made. The ABAC engine can make an authorization
decision when a subject requests access based on the assigned attributes of the requester, the assigned attributes of the object, environmental conditions, and a set of policies that are specified in terms of those attributes and conditions. This arrangement enables policies to be created and managed without direct reference to potentially numerous users and objects. Further, users and objects can be provisioned without reference to a policy [1].

ABAC allows us to specify fine-grained access control by combining various attributes of authorization elements into access control decisions. The attributes are predefined characteristics of subjects (e.g., job title and age), resources (e.g., data, programs, and networks), actions, and environments (e.g., current time and IP address). In ABAC, the subject presents a request to access the resource (object) [1]. The ABAC mechanism, then, evaluates policies, subject attributes, object attributes and environmental conditions to make the access control decision for the request. If authorized, the subject is given access to the resource. This is the core concept of ABAC which is illustrated in Figure 1.1.
XACML is an OASIS standard for ABAC policy specification. It provides both a policy language and an access control decision request/response language (both written in XML). The policy language is used to describe general access control requirements, and has standard extension points for defining new functions, data types, combining logic, etc. The request/response language is for defining a query to ask whether a given action should be allowed or not and interpreting the result. The response consists of a decision about whether the request should be permitted or not. The possible values in response are Permit, Deny, Indeterminate (an error occurred, or some required value was missing, so a decision cannot be made) or Non-applicable (the request can’t be answered by this service) [2].

The typical setup in an ABAC system is that someone (subject) wants to take some action on a resource (object). A subject will make a request to the system which is intercepted by a
Policy Enforcement Point (PEP) which is responsible for protecting the resources (like a filesystem or a web server). Once PEP receives a request, it will send the request to a Policy Decision Point (PDP) which will determine policies that are applicable to the request and determines whether access should be granted or not. To make an access decision, PDP needs to evaluate certain attributes for which it makes a query to the Policy Information Point (PIP) which is responsible for resolving the attribute values required to make access decisions. Once the PDP gets the required attributes, it makes access decisions from the applicable policies. The response (decision) is then returned to the PEP which will enforce the access decision [2]. Figure 1.2 depicts the data flow in an ABAC system. It also consists of a Policy Administration Point (PAP) which is responsible for management and administration of policies themselves.
XACML policies may have various faults due to the misunderstanding of access control requirements, the complexity of access control language, and coding errors. While ABAC is more expressive than traditional AC methods such as RBAC, however, it is complex, and its complexity increases the likelihood of the existence of faults resulting in vulnerabilities as well as the level of difficulty in revealing these vulnerabilities. Research has shown that XACML policies are subject to a variety of faults, such as incorrect rule targets, incorrect rule conditions, incorrect rule effects, incorrect targets of policies and policy sets, and incorrect uses of the rules or policy combining algorithms [3, 4].
A major means for finding faults in XACML policy is to execute an access control system with a test suite (a set of test cases). A test case includes test input (access request) and corresponding test Oracle (expected response). A test fails when the system’s actual response to the request is different from its expected response. Such failure often indicates the existence of a fault that may lead to unauthorized access or denial of service.

Current researches on testing XACML policies have commonly used policy mutation analysis to evaluate the fault detection ability of testing methods. It involves creating mutants of the policy under test using various mutation operators for a fault model (a set of the fault types). Each mutant is a variation of the given policy with an injected fault, which represents an error that a policy writer might make. A mutant is said to be killed if it fails one or more tests. The fault detection ability of the testing method is indicated by a mutation score which is the ratio between the number of mutants killed by the test suite and the total number of non-equivalent mutants.

Existing testing methods [5–12] vary in their overall fault detection ability, however, none of them can kill all the injected faults except for simple policies. The high-end ratios were usually obtained for simple policies and does not achieve such high-end ratios in complex policies. As a result, if these testing methods are applied in testing real-world access control systems with complex policy, it may not reveal many faults. As a result, it may leave significant vulnerabilities in the deployed system. In short, the existing approaches are inadequate for the high assurance of XACML-based access control. Moreover, it remains unclear what is essential to the fault detection given the fault and cost-effectiveness of testing various XACML testing methods.
To overcome afore-mentioned issues, we present a formalization of the fault detection conditions for illustrating what is essential for the fault detection of XACML policies in well-studied fault models [4,14,15] of XACML policies. The formalized fault detection condition is then used for formulating strong mutation-based test generation with the goal of achieving perfect mutation score. Finally, we have done the empirical study evaluating test suites from various coverage-based testing methods along with the test suites from strong mutation-based testing methods. The empirical study involves mutation analysis of various policies with different levels of complexity to identify fault detection capability and cost-effectiveness of each testing method.

1.2 Thesis Statement

The objective of this thesis is threefold. First, we formalize the fault detection conditions for faults in well-studied fault models of XACML policies. The goal is to make clear (formal) that what is essential for the fault detection given the fault. The fault detection condition of a given fault specifies the reachability, necessity, and propagation constraints that a test must satisfy in the order to reveal the fault. Although the notions of reachability, necessity, and propagation (a.k.a. sufficiency) constraints originate from mutation testing or constraint-based software testing [28,30], there is a lack of formal treatment of these constraints. In particular, the problem with propagation constraints of software is known to be intractable [29,30] because of the explosion of program execution paths. In this paper, the unique features of XACML make it feasible to formally represent the reachability, necessity, and propagation constraints of access control policies in three-valued logic.
Second, we formulate strong mutation-based test generation by exploiting the formalized fault detection condition to produce near-optimal test suites for XACML policies. A test suite for a given policy is said to be optimal if it contains the smallest number of tests which can achieve 100% mutation score. This paper considers near-optimality rather than strict optimality because constraints in XACML involve various data types, functions, and first-order predicates and solving the fault detection condition boils down to the constraint satisfaction problem, which is known to be undecidable. Nevertheless, due to the integration of all reachability, necessity, and propagation constraints for strong mutation testing, our approach is actually able to automatically generate the near-optimal test suites for all the XACML3.0 policies in the most recent literature [27]. This distinguishes our work from the existing mutation-based test generators that only deal with reachability and necessity constraints, a.k.a. weak mutation testing [20]. Generally, tests generated from weak mutation cannot achieve 100% mutation score [28, 31]. Further, SMT (strong mutation-based test generation) requires larger test generation time than all the testing methods discussed in this work. The objective of the SMT is to evaluate the cost-effectiveness and is not feasible for a policy with the larger number of the rules. Hence, we also presented NO-SMT (non-optimized strong mutation-based test generation) whose test generation time is greater than MC/DC but still comparable to it and is feasible to apply for larger policies.

Third, we present the quantitative evaluations of several test generation methods comparing against a near-optimal test suite (from SMT) of subject policies to establish cost-effectiveness and fault detection capability. The main testing methods for XACML
3.0 policies include rule coverage-based test generation, two forms of test generation with decision coverage, and two forms of MC/DC test generation [27].

1.3 Method

A fault in a policy is an error or flaw that causes it to produce an incorrect result. A fault may result in a different output than it is supposed to produce. We can exploit this difference in result between faulty policy and correct policy to reveal a fault. The idea is to record the output of the policy for some input when we know that policy is correct. Later, when we need to determine there exists fault or not, we supply those inputs to the suspect policy and if the result is different than the previously recorded result, we could conclude that there exists a fault. A fault policy, however, does not necessarily produce a different result than the correct policy for all possible inputs. The test inputs must satisfy certain constraint to produce a different result than from the correct policy and hence reveal the fault. Such constraints are referred to as fault detection conditions.

1.3.1 Fault Detection Condition (FDC)

The fault detection condition of a given fault specifies the constraints (or condition) that a test must satisfy to reveal the fault. A test input must satisfy the reachability, necessity and propagation condition to reveal the fault [18]. This is because to reveal the fault in a policy, the test must reach the faulty policy element which is termed as reachability condition. Once it is reached, the test must evaluate the faulty element to produce an incorrect intermediate result which is different than that from its correct counterpart which is referred to as necessity condition. If a test does not meet the reachability condition and/or necessity condition, the correct policy and faulty policy will behave the same and we do not have a means for detecting the fault. Once the incorrect intermediate result is produced,
it should play a role to produce a different result in the faulty policy than to the correct policy which is referred to as propagation condition. Hence, propagation condition is also essential for fault detection because reachability and necessity condition may only produce the incorrect intermediate result which may not be visible to the final access control decision if propagation condition is not met in which case we could not reveal the fault. Hence, we define fault detection condition with the following three constraints:

a) **Reachability**(*R*) constraint: the test must reach the faulty policy element (e.g., rule target, rule condition, rule effect, policy target, and combining algorithm).

b) **Necessity**(*N*) constraint: the test must make the faulty element evaluate to an incorrect intermediate result which is different from the evaluation result that should be produced by its correct counterpart.

c) **Propagation**(*P*) constraint: the test must make the faulty policy produce an incorrect response which is different from the expected response that should be produced by the correct policy.

We then exploit FDC to formulate strong mutation-based test generation.

### 1.3.2 Strong Mutation-based Test Generation with Fault Detection Conditions

Strong mutation-based test generation involves generating test input that satisfies the three constraints of the fault detection conditions - reachability, necessity and sufficiency/propagation for each fault type. If we use only reachability and necessity constraint to generate test suites, then it is called weak mutation-based test generation and if we use all three constraints, then it is referred to as a strong mutation. Strong mutation assures fault detection while weak mutation could not. However, incorporating propagation constraint is costly and requires a lot of effort that becomes infeasible to apply strong
mutation [20]. As a result, weak mutation is used in many cases, however, the features of XACML (being the domain specific language) made it feasible to apply strong mutation. In fault detection conditions, we specify all possible mutually exclusive constraints for reachability, necessity and propagation but for mutation-based test generation, we need not require generating test cases to satisfy all possible cases. For example, if reachability constraint is “policy target evaluates to true” or “policy target evaluates to an error”, we may just use “policy target evaluates to true” for fault detection and avoid “policy target evaluates to an error” to avoid redundant tests and for simplicity. Hence, we identify just sufficient mutually exclusive conditions from each of the reachability, necessity and propagation constraints for the fault to be detected. The process of identifying sufficient constraints to detect a fault involves picking one of the mutually exclusive constraints from the reachability constraint and concatenating the reachability constraint with corresponding mutually exclusive necessity constraints and propagation constraints. For example, if the reachability constraint has two mutually exclusive conditions which are “policy target evaluate to true” or “policy target evaluate to an error”, we can pick “policy target evaluates to true” as the reachability constraint. Further, the chosen reachability constraint may have two mutually exclusive necessity constraints as “rule target evaluates to true” or “rule target evaluates to an error”. Hence, we concatenate one of the mutually exclusive necessity constraints, say “rule evaluates to true”. Finally, we concatenate one of the mutually exclusive propagation constraints corresponding to the chosen necessity constraint, say it is “all rules with deny effect except the first rule should not evaluate to true”. Hence, the sufficient constraint to identify the fault is “policy target evaluates to true ∧ rule target should evaluate to true ∧ all deny rules except the first rule should not evaluate to true”.

The test suite generated in this way is called a mutation-based test suite. The resulting test suite may have many redundant test cases which will only kill those mutants which will be killed by other test cases. In other words, redundant test cases do not kill unique mutants that are not killed by any other test cases. Hence, such redundant test cases do not contribute to fault detection capability and result in poor cost-effectiveness of a test suite. Since our goal is to generate a near-optimal test suite, we need to optimize the test suite. Let, $M_i$ represents the set of mutants killed by an arbitrary test case $t_i$. By near-optimal, we mean, if $T = \{t_1, t_2, ..., t_n\}$ be a near-optimal test suite, then for any arbitrary $i$ and $j$, such that $i \neq j$, $M_i - M_j \neq \emptyset$ as well as $M_j - M_i \neq \emptyset$ i.e each test in a test suite kills at least one mutant not killed by any other test cases. The reason for generating a near-optimal test suite is that it can be used for evaluating the cost-effectiveness of a test suite from other testing methods. Hence, we have applied the optimization to find the near-optimal test suite.

Once we have the near-optimal test cases, we need Oracle value (expected response). Since we have the original policy which we assume to be correct, we run generated test inputs on original policy and record its value as Oracle values for a test suite. The optimized version of strong mutation-based test generation is named as SMT. The optimization involved in SMT is a costly operation making it infeasible to apply for large policies. We also formulate strong mutation-based test suite without optimization referred to as NO-SMT which has less cost-effectiveness as that of SMT but achieves perfect mutation score.
1.3.3 Quantitative Analysis

Finally, we do quantitative analysis using metrics like mutation score and mutants killed per test (MKPT) to determine the cost-effectiveness of major testing methods of XACML (such as rule coverage, decision coverage, non-error decision coverage, MC/DC coverage and non-error MC/DC coverage). Mutation score is the percentage of mutants killed against the number of non-equivalent mutants. It indicates the fault detection capability of a test method. MKPT is the average number of mutants killed by a test in a test suite. An optimal test suite will have the highest MKPT score. As a result, we can evaluate the cost-effectiveness of a testing method by comparing the MKPT score of the test suite from the method under consideration with an optimal test suite. However, finding an optimal test suite is an undecidable problem and hence, we used the SMT with the approximation method to optimize the test suite to find near-optimal test suite. The goal is to compare the MKPT score of a near-optimal test suite with the test suite from the current method and establish the cost-effectiveness of the testing method.

The rule coverage is the coverage criteria which aims to evaluate the effect of each rule in the policy. The decision coverage is the coverage criteria which aims to evaluate each decision point (policy set target, policy target, rule target and rule condition) to three possible evaluations true, Non-applicable (N/A) and error. Non-error decision coverage is the same as decision coverage except that it does not consider the error in evaluation of decision points i.e. it aims to evaluate each decision point to only true and N/A [27]. Consider, an expression “resource-id=Liquor ∨ resource-id= Medicine” which is composed of a disjunction of two constraints “resource-id=Liquor” and “resource-id= Medicine”. A coverage criterion which satisfies decision coverage and in addition requires
that every condition in a decision point has taken on all possible outcomes at least once and each condition has been shown to independently affect the decision’s outcome is referred to as MC/DC coverage criteria. For example, MC/DC of a conjunctive expression with n conditions (e.g., $c_1 \land \ldots \land c_n$) requires $n+1$ tests: one test that evaluates all conditions to true and n tests that evaluate one condition to false and other conditions evaluate to true. MC/DC of a disjunctive expression with n conditions (e.g., $c_1 \lor \ldots \lor c_n$) requires $n+1$ tests: one test that evaluates all conditions to false and n tests that evaluate one condition to true and other conditions evaluate to false [27].

1.4 Outline

The remainder of this document is organized as follows. Chapter 2 presents the background and summarizes related work on the research topic. Chapter 3 specifies the Fault Detection Condition (FDC) for the fault model. Chapter 4 presents the mutation-based test generation, Chapter 5 specifies the quantitative evaluation, and Chapter 6 concludes this work.
CHAPTER 2

Background and Related Work

2.1 Mutation analysis and mutation-based test generation

Mutation analysis is a fault-based testing technique which provides a testing criterion that can be used to measure the effectiveness of a test set. The general principle of Mutation analysis is to produce mutants of the original source/specification by injecting the mistakes that programmers/users might make. Such faulty programs/specification resulted after deliberately seeding faults into the original source are called mutants. The statement in which mutation takes place is called a mutation point and the transformation rules used to produce such fault and hence mutants are called mutation operators. The resulting faulty policies (mutants) which exhibit the same behavior as the original ones are known as equivalent mutants and those which exhibit different behavior than that of the original ones are called nonequivalent mutants [19].

Mutation operators are defined with respect to a fault model, which is a collection of the fault types in the programming language. The main hypotheses of mutation testing [19, 32] include: (a) the mutants are based on actual fault models and are representative of real faults, (b) developers produce programs (policies) that are close to being correct, (c) tests sufficient to detect simple faults (i.e., in mutants) are also capable of detecting complex faults. Experiments have shown that mutants are indeed similar to real faults for the purpose of evaluating testing techniques [16, 33]. To assess the quality of a test set, the
generated mutants are run against the test set. If the result from a mutant is different from the original one for any test cases in the input test set, we say that the mutant failed the test. A mutant is said to be killed if it fails one or more tests. One outcome of the mutation testing process is the mutation score – mutant killing ratio. Mutant-killing ratio is the ratio between the number of mutants killed by the test suite against the total number of non-equivalent mutants [3].

Mutation-based test generation derives a test from one or more mutants of a given program so that the mutant and its original program produce a different execution result. Such a test needs to meet the following constraints: (a) Reachability constraint: the test must reach the mutation point, i.e., trigger the execution of the mutated code, (b) Necessity constraint: the test must make the mutated code evaluate to an intermediate result that is different from that of the original program, and (c) Propagation (a.k.a. sufficiency) constraint: the test must make the intermediate result of the mutated code propagate to a final state that is different from the final state of the original program. We refer to the collection of reachability, necessity, and propagation constraints as the fault detection condition. The existing techniques primarily follow the concept of weak mutation testing [20] that uses the reachability and necessity constraints to generate test inputs [31]. The main reason is that it has been shown to be intractable to solve the propagation constraint [28]. This paper aims at strong mutation testing of XACML policies that deal with all reachability, necessity, and propagation constraints. As a domain-specific language, XACML has a unique structure that makes it feasible to tackle propagation constraints. The existing work on mutation testing of XACML policies focuses on mutation tools and
evaluation of testing methods with policy mutants. In comparison, this paper focuses on the formalization of the fault detection conditions and mutation-based test generation.

2.2 Introduction to XACML

XACML is a general-purpose access control policy language. The root of an XACML policy document $\mathcal{P}$ is a policy element or a policy set element. A policy set element contains other child policy elements or policy set elements such that each of which may evaluate to different access control decisions. A policy element contains a list of the rule elements each of which may evaluate to different access control decisions. And, there is a mechanism to resolve decisions from multiple units of the rules within a policy or multiple units of policies or policy sets within a policy set known as combining algorithms. The combining algorithm which reconciles decisions from a list of the rules is referred to as rule combining algorithm. Similarly, the combining algorithm which reconciles decisions from policies or policy sets is referred to as policy combining algorithm. A policy set, policy or rule element contains a target element that specifies the set of requests to which it applies. Further, a rule may consist of another Boolean function known as condition element which needs to be satisfied for a rule to be applied. Figure 2.1 depicts the XACML policy language model.
As shown in Figure 2.1, the target of a rule, policy, or policy set is a conjunctive sequence of $AnyOf$ clauses. Each $AnyOf$ clause is a disjunctive sequence of $AllOf$ clauses, and each $AllOf$ clause is a conjunctive sequence of match predicates. A match element matches and compares attributes in a request context with the embedded attribute values. Logical expressions for match predicates and rule conditions are usually defined on four categories of attributes: subject, resource, action, and environment. They can use a great variety of predefined functions and data types. A rule also has an effect element which will be either permit or deny corresponding to the access decision of the rule.

Formally, a policy set element $PS$ is a quintuple $< PST, PCA, [P_1, P_2, \ldots, P_m], A, O >$, where $PST$ is the policy set target, $PCA$ is the policy combining algorithm, and $[P_1, P_2, \ldots, P_m]$ is the list of policies or policy sets in the policy set, $A$ is a set of advice, and $O$ is a set of Obligation. Each policy $P_i$ is a quintuple $< PT_i, RCA_i, [r_{i1}, r_{i2}, \ldots, r_{im}], A_i, O_i >$, where $PT_i$ is the policy target, $RCA_i$ is the rule combining algorithm, and $[r_{i1}, r_{i2}, \ldots, r_{im}]$ is the list of 

![Figure 2.1: XACML policy language model [2]](image-url)
of the rules in the policy, $A_i$ is a set of advice, and $O_i$ is a set of obligation. Each rule $R_j$ is a triple $<rt_j, rc_j, re_j>$, where $rt_j$ is the rule target, $rc_j$ is the rule condition, and $re_j \in \{permit, deny\}$ is the rule effect. $<rt_j, rc_j, permit>$ is called a permit rule, whereas $<rt_j, rc_j, deny>$ is a deny rule. If both $rt_j$ and $rc_j$ are omitted (always true), then the rule $<_, _, re_j>$ is a default rule. More specifically, $<_, _, deny>$ is a default deny rule, whereas $<_, _, permit>$ is a default permit rule.

To access the resource, a subject presents an access request to the system. An access request for an ABAC authorization system consists of a set of attributes. For an access request $q$, a policy or policy set responds with an access decision, such as permit or deny. Given an access request $q$, $PS$ is evaluated to produce a response (i.e., access decision) denoted as $d(PS, q)$. A policy set target $PST$ is first evaluated according to the attribute values in $q$. If the result of the evaluation is $false$, then $d(PS, q) = N/A$ otherwise policies $P_1, P_2, \ldots, P_m$ will be evaluated if $PST$ is $true$ or evaluates to an error. $d(PS, q)$ depends on policy combining algorithm $PCA$ and the decisions of individual policies with respect to $q$ (denoted as $d(P_i, q)$). Similarly, for an individual policy $P_i = <PT_i, RCA_i, [r_1, r_2, \ldots, r_n]>$, policy target $PT_i$ is evaluated according to the attribute values in $q$. If the evaluation result is $false$, then $d(P_i, q) = N/A$, otherwise, rules $r_1, r_2, \ldots, r_n$ will be evaluated. $d(P_i, q)$ depends on rule combining algorithm $RCA_i$ and the decisions of individual rules. Decision of the rule $r_j = <rt_j, rc_j, re_j>$ with respect to $q$, denoted as $d(r_j, q)$, is defined as follows:

i) $Permit$: access is granted when $re_j = permit$ and $rt_j$ and $rc_j$ is $true$ with respect to $q$.  

ii) $Deny$: access is denied when $re_j = deny$, and $rt_j$ and $rc_j$ is $true$ with respect to $q$.  

iii) Non-applicable, or simply N/A: $q$ is not applicable, i.e., $rt_j$ and/or $rc_j$ is false with respect to $q$.

iv) IndeterminateD or simply $I(D)$: An error occurred when $rt_j$ or $rc_j$ was evaluated and $re_j = Deny$. The decision could have evaluated to Deny if no error had occurred. A syntactically valid access request may cause the occurrence of a runtime error for different reasons, such as missing an attribute value, mismatch of an attribute type, and an exception of expression and function evaluation.

v) IndeterminateP or simply $I(P)$: An error occurred when $rt_j$ or $rc_j$ was evaluated and $re_j = Permit$. The decision could have evaluated to Permit if no error had occurred.

A rule may have an empty target as well as empty conditions referred to as a default rule. For a default rule $r_j = < _, _, re_j >$, any access request $q$ is $d(r_j, q) = re_j$.

The root element of a general XACML policy document $\mathbb{P}$ could be either a policy element or a policy sets element. If the root element is a policy set element, then policy combining algorithms for the root policy set has nested rule combining algorithms and/or policy combining algorithms inside it. Since we need to deal with five rule combining algorithms in our work and if we consider policy sets we need to consider six policy combining algorithms, such nesting would create lots of combinations of nested combining algorithms. As a result, for simplicity, further, in this work, we would only consider XACML policy document $\mathbb{P}$ which has a policy element as a root of the XACML document so that we don’t need to deal with nested rule and policy combining algorithms. With similar reasoning, we can also deal with policy sets and hence policy combining algorithms but for simplicity, we only consider policy in this work. Further, Advice and Obligation plays no role in our work, so we omit them while representing policy elements further in
this work. Hence, for simplicity, we represent XACML policy document \( \mathcal{P} \) as 
\[
\mathcal{P} = \langle \mathcal{PT}, \mathcal{RCA}, [r_1, r_2, ..., r_n] \rangle
\]
where \( \mathcal{PT} \) is policy target, \( \mathcal{RCA} \) is rule combining algorithm and the list of child rules of \( \mathcal{P} \).

### 2.2.1 Sample Policy

Figure 2.2 presents an example of an XACML policy document which has Policy Id \( \text{KmarketBluePolicy} \). It is a demonstration policy from Balana – Open source implementation of XACML [13]. The rule combining algorithm of the policy is \textit{deny-overrides} (line 2). The policy’s target (lines 3-14) implies the constraint “\( \text{role}=\text{blue} \)” where a role is an attribute in the subject category and blue is the value for the attribute of type string. For this policy to be applied to a request, the request context must contain the subject attribute role with the value of blue.

There are three rules with rule ids: \text{deny-liquor-medicine} (line 16-37), \text{max-drink-amount} (lines 38-61), and \text{permit-rule} (line 62). The target of the rule \text{deny-liquor-medicine} (lines 18-36) implies the constraint “\( \text{resource-id=Liquor} \)” (line 19-26) \( \lor \) “\( \text{resource-id=Medicine} \)” (lines 27-34), where resource-id is an attribute in the resource category. Since the rule does not have a condition element, it is \textit{true} by default, hence, the rule will result in a “Deny” decision if “\( \text{resource-id=Liquor} \lor \text{resource-id=Medicine} \)”.

The target of the rule \text{max-drink-amount} implies the constraint “\( \text{resource-id=Drink} \)” and the condition implies constraint “\( \text{amount} > 10 \)”.

Thus, the rule results in a deny decision if “\( \text{resource-id=Drink} \land \text{amount} > 10 \)”.

Rule \text{permit-rule} has neither target nor condition. It results in a Permit decision whenever it is reached.
Figure 2.2. A sample XACML policy [13]
Hence, in the notation we described earlier, $P = \langle role = \text{"blue"}, \text{deny-overrides} \rangle, [r_1, r_2, r_3]$ such that $r_1 = \langle rt_1, rc_1, re_1 \rangle$, $r_2 = \langle rt_2, rc_2, re_2 \rangle$, $r_3 = \langle rt_3, rc_3, re_3 \rangle$ such that $rt_1 = \text{"resource-id = Liquor } \lor \text{ resource-id = Medicine"}$, $rc_1 = \text{true}$ (since its empty), $re_1 = \text{deny}$, $rt_2 = \text{"resource-id = Drink"}$, $rc_2 = \text{"amount > 10"}$, $re_2 = \text{Deny}$, $rt_3 = \text{true}$ (since its empty), $rc_3 = \text{true}$ (since its empty), and $re_3 = \text{Permit}$.

To illustrate how an XACML authorization scheme works, we first need to discuss how each element of XACML evaluation occurs.

### 2.2.2 Policy Evaluation

When a request is presented to the AC system, it first needs to determine whether the available set of XACML policies can be applied to a given request or not. For this purpose, the target element $PT$ of root policy element $P$ of XACML policy document $ℙ$ is used. If it is empty, the policy is applicable to any request $q$. If it is not empty and request $q$ meets the constraints specified by the target $PT$ or if there is an error while evaluating the target $PT$, the policy is applicable to request $q$ otherwise policy is not applicable and will not be evaluated further. The evaluation of policy involves evaluation of its child rules. Once its child elements are evaluated and it obtains the authorization decision from each child element, it uses a rule combining algorithm to reconcile the decision obtained from various child rules and makes a final authorization decision from the result of the rule combining algorithm [2]. The details on combining algorithms is presented in Section 2.2.5. If policy target $PT$ evaluates to $\text{true}$, the policy-level decision will be the decision from the rule combining algorithm itself. However, if $PT$ evaluates to an error, the decision will be made as described in Section 2.2.3.
2.2.3 Policy Evaluation and Indeterminate Target

If the target of a policy evaluates to an error, the result of policy evaluation will be N/A whenever the result of the rule combining algorithm is N/A; the result will be $I(P)$ whenever the result of the rule combining algorithm is permit; the result will be $I(D)$ if the result of the rule combining algorithm is deny, and the result will be $I(DP)$ if result of combining algorithm is Indeterminate. For any other indeterminate result {$I(DP), I(D), I(P)$} from the rule combining algorithm, the result for policy evaluation is the same as that for the rule combining algorithm. It uses a rule combining algorithm to reconcile decisions obtained from various child rules and makes a final authorization decision from the result of the rule combining algorithm [2].

2.2.4 Rule Evaluation

The evaluation of $i^{th}$ rule $r_i$ occurs if the evaluation of the rules above it does not halt the evaluation of policy $P$ producing the result $d(P,q)$. The rule $r_i$ is applicable to request $q$ if “rule target $rt_i$, as well as rule condition $rc_i$, is true” or “rule target $rt_i$ evaluates to an error” or “rule target $rt_i$ is true and rule condition evaluates to an error” [2]. We say rule evaluates to true, if rule target, as well as rule condition, evaluates to true. Similarly, if rule target or rule condition evaluates to false, we say rule evaluates to false or N/A. If rule target or rule condition evaluates to an error, we say rule evaluates to an error.

If a rule evaluates to true, the rule-level decision ($d(P,r_i)$) will be the effect of the rule i.e if the effect of the rule is permit, then the decision of the rule will be permit. If a rule evaluates to an error, the rule-level decision will be the Indeterminate of corresponding effect i.e if the effect is permit, the decision of the rule will be $I(P)$ or simply $I(P)$. 
For a request $q$, there may be multiple rules which are applicable such that each produces their effect and it is the job of the rule combining algorithm (RCA) to reconcile decision of multiple rules and produce the final decision for the policy evaluation. Section 2.2.5 discusses various rule combining algorithms we considered in this work.

Let $rca(P, q)$ denote the result of applying RCA to the rules in $P$ for the request $q$. Assuming that the list of the rules in $P$ is non-empty, $rca(P, q) \in \{\text{Permit}, \text{Deny}, \text{N/A}, \text{I(D)}, \text{I(P)}, \text{I(DP)}\}$ and per the standard specification [2], $d(P, q)$ is defined as follows:

$$d(P, q) = \begin{cases} 
\text{N/A}, & \neg PT \\
\text{PT} & rca(P, q) \\
\text{Error(PT)} & \land rca(P, q) \in \{\text{N/A, I(D)}, \text{I(P)}, \text{I(DP)}\} \\
\text{I(P)}, & \text{Error(PT)} \land rca(P, q) = \text{Permit} \\
\text{I(D)}, & \text{Error(PT)} \land rca(P, q) = \text{Deny} 
\end{cases}$$

2.2.5 Combining algorithms

There are eleven RCAs in XACML 3.0. Four of them are for compatibility support for older versions - legacy ordered-deny-overrides, legacy deny-overrides, legacy ordered-permit-overrides, and Legacy ordered-permit-overrides [2]. As they are for the backward compatibility for the previous version of XACML, we do not consider them in our work.

Among the remaining seven, the five are listed below and the other two are ordered deny-overrides and ordered permit-overrides. In Balana [13] (an open source implementation of XACML3.0), the implementations of ordered-deny-overrides and ordered-permit-overrides are the same as deny-overrides and Permit-overrides. As a result, we do not consider these two as well and this work only focuses on the five RCAs which are as follows:-

a) deny-overrides: deny-overrides is intended for those cases where a deny decision should have priority over a Permit decision. If any rule evaluates to deny, the result is deny. If there is no deny decision from any rules and if any decision is I(DP), the result is I(DP).
If there is no deny and I(DP) decision, and if any decision is I(D) and another decision is I(P) or deny, the result is I(DP). If it is also not true, then if any decision is I(D), the result is I(D). If there is no deny, I(DP) and I(D) decision, and if any decision is permit, the result is permit. If it is also not true and if any decision is I(P), the result is "I(P)" otherwise, the result is N/A [2].

b) permit-override: permit-override RCA is intended for those cases where a permit decision should have priority over a deny decision. If any decision is permit, the result is permit. If there is no permit decision and if any decision is I(DP), the result is I(DP). If there is no permit and I(DP) decision, and if any decision is I(P) and another decision is I(D) or deny, the result is I(DP). If it is also not true, then if any decision is I(P), the result is I(P). If there is no permit, I(DP) and I(P) decision, and if any decision is deny, the result is deny. If it is also not true and if any decision is I(D), the result is I(D) otherwise, the result is N/A.

c) deny-unless-permit: This is intended for those cases where a permit decision should have priority over a deny decision, and an Indeterminate or N/A must never be the result if the policy is applicable to the request. If any decision is permit, the result is permit otherwise, the result is deny.

d) permit-unless-deny: This RCA is intended for those cases where a deny decision should have priority over a permit decision, and an Indeterminate or N/A must never be the result. If any decision is deny, the result is deny else the result is permit.

e) first-applicable: This RCA is intended for those cases where the evaluation of policy should halt as soon as any rule is applicable to request q. Rules are evaluated in the order in which they are listed. If a rule’s target matches and condition evaluates to true,
then the result is rule’s effect (permit or deny). If a rule’s target evaluates to an error or rule condition evaluates to an error, then the result is \( I(P) \) if rule’s effect is permit or \( I(D) \) if rule’s effect is deny. If the target or condition evaluates to false, the next rule is evaluated. If no further rule exists, then the result is N/A.

To illustrate how XACML policy evaluation occurs, the following section presents sample requests and discusses how the policy evaluation takes place for the given sample policy.

### 2.3 Sample Requests and Policy Evaluation

This section presents some sample requests and how the evaluation occurs in sample policy. The XACML request contains a list of attributes and their value. Let us consider q1 be the first sample request as \( q_1 = \{ \text{resource-id} = "\text{Liquor}" \} \). This request consists of one attribute-value pair where resource-id is attribute and Liquor is a value of the attribute. Since the policy target of sample policy is “role = blue” and \( q_1 \) does not contain role attribute, the given sample policy will not be applicable to \( q_1 \) because the policy target did not match, and no rules will be evaluated, and the final decision will be N/A.

Let us consider second request \( q_2 \) as \( \{ \text{resource-id} = "\text{Liquor}" \text{ and role} = "\text{gold}" \} \). Since \( q_2 \) does contain role attribute but it is not blue, the policy will not be applicable because the policy target did not match, and the final decision of the policy evaluation will be N/A.

Let us consider a third request \( q_3 \) as \( \{ \text{resource-id} = "\text{Liquor}" \text{ and role} = "\text{blue}" \} \). Since \( q_3 \) does contain role attribute and is blue the policy will be applicable. Further, it also consists of another attribute-value pair resource-id = “Liquor”. As a result, rule 1 is applicable whose effect is deny and since the RCA is deny-overrides, the rule evaluation
stops after rule 1 and decision of RCA will be deny. Since the PT evaluates to true, the final
decision for the policy evaluation will be deny. However, if PT evaluated to an error, the
final decision for policy would be I(D).

Let us consider a fourth request q4 as \{resource-id = "aFFFF" and role = "blue"\}. Since role attribute is blue, the policy will be applicable. The value of another attribute
resource-id is aFFFF. As a result, rule 1 is not applicable as well as rule 2 because their
target didn’t match. However, rule 3 is a default rule with no target and condition, so it will
be applicable to any request and produces permit effect. Since the rule level decision of the
first two rules are N/A and that of the third rule is permit, the RCA level decision will be
permit. Since PT evaluates to true, the final policy-level decision will be permit.

2.4 Related Work

A test for an XACML policy consists of a test input and the corresponding Oracle
value (i.e., expected response to the access request). Oracle values depend on the access
control requirements of the system under test. A test fails when the system’s actual
response to the request is different from the expected response. Such a failure often
indicates the existence of a fault that may lead to unauthorized access, elevated privilege,
or denial of service. The existing approaches to test generation for XACML policies fall
into two categories: model-based testing that derives tests from models, and policy-based
testing that produces test inputs directly from the policy under test. As access control
policies are extra-constraints on system functions, the model-based testing approach
usually integrates functional models with access control specifications and can generate
both test inputs and Oracle values. This paper is mostly related to the work that generates
test inputs from the XACML policy under test.
The existing testing methods for XACML policies generate access requests directly from the policy under test. A user needs to define the expected response for each request to determine whether each test passes or fails. Martin et. al. generates access requests in Cirg from counterexamples produced by model checker Margrave [21] through the change-impact analysis [10]. Mutation score of the testing methods in Cirg ranged from 30% to 60% in different case studies and 100% for a simple policy. Targen [22] obtained mutation score that ranged from 75% to 79% for different case studies which derive access requests to satisfy all the possible combinations of truth-values of the attribute id-value pairs found in a given policy [5]. The X-CREATE framework deals with the structures of the Context Schema Considering that requests must conform to the XML Context Schema. Bertolino et al. have developed the Mutant-killing ratios of the X-CREATE framework ranging from 75% to 96% for several small policies [7]. They have also developed other test selection strategies, such as Simple Combinatorial and Incremental XPT [6].

Mutant-killing ratios of the Simple Combinatorial strategy ranged from 3% to 100%, whereas mutant killing ratios of the Incremental XPT strategy ranged from 55% to 100%. Bertolino et al. [8] proposed an approach to selecting tests based on the rule coverage criterion. It chooses existing tests to match each rule target set, which is the union of the target of the rule and all enclosing policy and policy set targets. Mutant-killing ratios of this approach ranged from 62% to 98%. In addition, Bertolino et al. [9] proposed similarity-based metrics for prioritizing existing tests of policies. This work is not concerned with how the tests are generated, though. Li et al. [12] have developed XPTester, which used symbolic execution technique to generate requests from XACML policies. They convert the policy under test into semantically equivalent C Code Representation (CCR) and
symbolically execute CCR to create test inputs and translate the test inputs to access requests. Mutant-killing ratios of XPTester ranged from 37% to 93%. Although all the above work uses mutation to evaluate fault detection ability, there are subtle differences between the subject policies and the fault models used by different research groups. It is obvious that the above methods are far from satisfactory for the high assurance of XACML policies. Most of them produce many tests by combining attribute values. None of them have considered advanced coverage criteria, e.g., decision coverage and MC/DC, for access control constraints (i.e., rule target, rule condition, policy target, and policy set target).

Verification techniques have also been proposed for quality assurance of XACML policies. The verification system in Margrave checks whether an XACML policy satisfies given properties that describe the constraints on attributes. Margrave transforms the XACML policy into multi-terminal binary decision diagrams. Hwang et. al. [23] applied Margrave to the detection of multiple-duty-related security leakage. Hughes and Bultan [25] developed an approach for defining properties as partial orderings between XACML policies, translating them to Boolean formulas, and using the Zchaff SAT solver to check satisfiability of the Boolean formulas. Hughes and Bultan have also proposed an approach for translating XACML policies into the Alloy language and analyzing properties as partial ordering relations [24]. The above verification techniques are premature because they only deal with a very restricted subset of XACML (e.g., no or limited attribute data types and no complex conditionals). In addition, they require formal representation of application-specific properties, which can be a non-trivial task for XACML users.
CHAPTER 3

Fault Detection Condition

A fault in a policy is an error or flaw that can make it produce an incorrect result. Consider, the \textit{i}^{th} rule of a policy \textit{P} should have permit effect, but if it is somehow changed to deny, then there is a fault in the effect of a rule. Since the effect of a rule in a policy is incorrect, we refer to such a fault as an Incorrect Rule Effect fault. Similarly, if there is a fault in the target element of a rule, then it is referred to as an Incorrect Rule Target fault. Incorrect Rule Condition fault represents the fault in the condition element. If a rule is missing, we refer to such a fault as a missing rule fault. If there is a different combining algorithm than it is supposed to have, then it is an called Incorrect Rule (Policy) Combining Algorithm fault.

Table 3.1: Fault Model

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Meaning</th>
<th>Fault Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CRE</td>
<td>Change Rule Effect</td>
<td>Incorrect Rule Effect</td>
</tr>
<tr>
<td>2</td>
<td>RTT</td>
<td>set Rule Target True</td>
<td>Incorrect Rule Target</td>
</tr>
<tr>
<td>3</td>
<td>RTF</td>
<td>set Rule Target False</td>
<td>Incorrect Rule Condition</td>
</tr>
<tr>
<td>4</td>
<td>RCT</td>
<td>set Rule Condition True</td>
<td>Incorrect Rule Condition</td>
</tr>
<tr>
<td>5</td>
<td>RCF</td>
<td>set Rule Condition False</td>
<td>Incorrect Rule Condition</td>
</tr>
<tr>
<td>No.</td>
<td>Code</td>
<td>Description</td>
<td>Category</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>6</td>
<td>ANF</td>
<td>Add Not Function in condition</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>RNF</td>
<td>Remove Not Function in condition</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>RER</td>
<td>Remove a Rule</td>
<td>Missing Rule</td>
</tr>
<tr>
<td>9</td>
<td>FPR</td>
<td>First Permit Rule</td>
<td>Incorrect Rule Ordering</td>
</tr>
<tr>
<td>10</td>
<td>FDR</td>
<td>First Deny Rule</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>PTT</td>
<td>set Policy Target True</td>
<td>Incorrect Policy Target</td>
</tr>
<tr>
<td>12</td>
<td>PTF</td>
<td>set Policy Target False</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>RPTE</td>
<td>Remove Parallel Target Element</td>
<td>Missing target element</td>
</tr>
<tr>
<td>14</td>
<td>CRC</td>
<td>Change Rule Combining Algorithm</td>
<td>Incorrect Combining Algorithm</td>
</tr>
</tbody>
</table>

Similarly, if there is a fault in the ordering of the rules, then it is referred to as an Incorrect Rule Ordering fault. If either Match element, AnyOf element or AllOf element of Target element is missing, we refer to such a fault as a Missing Parallel Target Element fault. If there is a fault in policy target, then we refer to such a fault as an Incorrect Policy Target fault.

While defining policy, there could exist various faults. The policy with the fault is referred to as a faulty policy. If P’ is a fault policy of original policy P, then P’ is nothing but the result of an application of some form of transformation rule that introduces the fault. For example, the transformation rule could be “change rule effect” which alters the
effect of the rule from permit to deny and vice versa. In mutation analysis literature, the transformation rule which results in the faulty policy from the correct one is referred to as a mutation operator and the faulty policy itself is referred to as a mutant. Table 3.1 consists of fourteen mutation operators which are categorized into eight faults types. Mutation operators are defined with respect to a fault model, which is a collection of the fault types in the given domain of programing language or specification. The details on each mutation operator in Table 3.1 are discussed in Sections 3.1 through 3.14.

The fault in a policy may result in it to produce an incorrect result i.e. it results in a different output than it is supposed to produce. We can exploit this difference in result to reveal a fault by supplying the same input to both original policy and mutant, and if there is a difference in response, then we could conclude that there exists a fault. A fault policy, however, does not necessarily produce a different result than the correct policy for all inputs. The test inputs must satisfy certain conditions to produce a different result than from the correct policy and hence reveal the fault. Such conditions are referred to as fault detection conditions.

Hence, the fault detection condition (FDC) of a given fault should specify the constraints that a test case must satisfy to reveal the fault. As discussed in Section 1.3.1, the fault detection condition can be formulated with three constraints which are reachability constraint, necessity constraint and propagation constraint.

a)  *Reachability(R) constraint:* Reachability constraint specifies that the test must reach the faulty policy element (e.g., rule target, rule condition, rule effect, policy target, and combining algorithm). If the faulty element is not evaluated, then the faulty policy will behave the same as that of the original policy and we cannot distinguish the fault. The first
rule of a sample policy in Figure 2.2 of Chapter 2 has deny effect with id “deny-liquor-medicine”. Consider there is a faulty policy which is same as the original policy except for the effect of the first rule with id “deny-liquor-medicine” has permit effect instead. To identify this fault, this rule must be evaluated. To evaluate the rule, the rule should be reached during policy evaluation. To reach the first rule, the policy target must evaluate to \( \text{true} \) or \( \text{error} \). Hence, the reachability constraint here is to evaluate policy target to \( \text{true} \) or \( \text{error} \).

\( b) \) \( \text{Necessity}(N) \) constraint: The test must make the faulty element evaluate to an incorrect intermediate result, which is different from the evaluation result that should be produced by its correct counterpart. For example, for the fault considered in reachability constraint above, the rule target and/or condition of the faulty rule should evaluate to an error or \( \text{true} \) so that it produces an incorrect intermediate result. Hence, the necessity constraint is “rule target and rule condition of the faulty rule does not evaluate to \( \text{N/A} \)” (i.e. either rule target and condition evaluates to \( \text{true} \) or rule target evaluates to an error or rule target evaluates to \( \text{true} \) and rule condition evaluates to an error).

\( c) \) \( \text{Propagation}(P) \) constraint: The test must make the faulty policy produce an incorrect response, which is different from the expected response that should be produced by the correct policy. For example, to propagate incorrect intermediate results for revealing the fault specified in reachability constraint, all the other rules with deny effect except the first rule should not evaluate to \( \text{true} \) i.e. either they should evaluate to an error or \( \text{false} \) because the rule combining algorithm is \( \text{deny-overrides} \) and if any another deny rule evaluates to \( \text{true} \), it will produce the deny result in both faulty and correct policy.
We use the notation $PT$, $\neg PT$ and $Error(PT)$ to denote that policy target evaluates to $true$, $N/A$ and $error$ respectively. We mention rule evaluates to $true$ if both target and condition of the rule evaluates to $true$. We use the expression $rb_i = (rt_i \land rc_i)$ to denote that the target, as well as condition of $i^{th}$ rule, evaluates to $true$. Similarly, $Error(rb_i)$ denotes an $error$ in the evaluation of the target or condition of the rule $r_i$. Further, $\neg rb_i$ denotes either $rt_i$ or $rc_i$ evaluates to $false$ such that rule is $N/A$. Additionally, if none of the attributes in the request context matches the attributes in the rule target or rule condition, then the rule is not applicable for the request. We use the notation $I(P)$, $I(D)$ and $I(DP)$ to denote $Indt$, $I(P)$ and $Indeterminate$ respectively. We use the notation $r_i = <rb_i, re_i>$ whenever possible to represent $i^{th}$ rule $<rt_i, rc_i, re_i>$ for simplicity. Further, we interchangeably use notation $r_i$ or $current$ rule $under$ $consideration$ to denote an arbitrary $i^{th}$ rule.

3.1 FDC for Change Rule Effect (CRE)

Change Rule Effect (CRE) is a mutation operator for the incorrect rule effect fault type in which there is a fault in effect of a rule element. Since there are only two possible rule effects - permit or deny, there will be a fault in effect of a rule if the effect of a rule gets altered from permit to deny and vice-versa. The flipping of the rule effect is the only mutation operator for incorrect rule effect fault.

Consider an XACML policy $P = <PT, RCA, RL>$ where $PT$ is Policy Target, $RCA$ is Rule Combining Algorithm and $RL = [r_1, r_2, \ldots, r_n]$ is a list of the rules. If the effect of $i^{th}$ rule $r_i$ is flipped to deny by some incident, then the resulting policy will be $P'$ as shown in Table 3.2. Since $i^{th}$ rule is supposed to have permit effect but in $P'$ it is $deny$, so $P'$ is the faulty policy. Here, $P'$ is called the Change Rule Effect (CRE) mutant of $P$. 
Table 3.2: A Faulty Policy with an Incorrect Rule Effect

<table>
<thead>
<tr>
<th></th>
<th>Correct Policy $P$</th>
<th>Faulty Policy $P'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy target</td>
<td>$PT$</td>
<td>$PT$</td>
</tr>
<tr>
<td>Rule combining algorithm</td>
<td>Permit-Overrides</td>
<td>Permit-Overrides</td>
</tr>
<tr>
<td>Rules $R$</td>
<td>$r_1$</td>
<td>$r_1$</td>
</tr>
<tr>
<td></td>
<td>$...$</td>
<td>$...$</td>
</tr>
<tr>
<td></td>
<td>$r_i$</td>
<td>$r_i$</td>
</tr>
<tr>
<td></td>
<td>$&lt;rb_i, re_i&gt;$</td>
<td>$&lt;rb_i, re_i&gt;$</td>
</tr>
<tr>
<td></td>
<td>$...$</td>
<td>$...$</td>
</tr>
<tr>
<td></td>
<td>$r_n$</td>
<td>$r_n$</td>
</tr>
<tr>
<td></td>
<td>$&lt;rb_n, re_n&gt;$</td>
<td>$&lt;rb_n, re_n&gt;$</td>
</tr>
</tbody>
</table>

Since the fault detection condition depends on the rule combining algorithm (RCA) of a policy, we present fault detection condition for each of the RCA.

a) Permit-overrides:

i) Reachability constraint

The reachability constraint must trigger the evaluation of the rule with faulty effect i.e. it should result in the evaluation of $i^{th}$ rule in both $P$ and $P'$. The rules in a policy will only be evaluated if the policy target is true or evaluates to an error. Further, when the rule-combining algorithm is Permit-overrides, rule $r_i$ will not be triggered if there is a permit rule before rule $r_i$ that evaluates to a permit decision. Thus, the reachability constraint is - for any permit rule $r_j$ ($j<i$) before rule $r_i$, $rb_j$ should not evaluate to true (i.e. it should be N/A or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to true or error and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A or error (i.e should not evaluate to true).
In formal notation, it will be \{PT \lor Error(PT)\} \land \{ \neg rb_j \lor Error(rb_j) \} for any permit rule \( r_j = <rb_j, \text{permit}> \) for \( j < i \)

ii) **Necessity constraint**

The necessity constraint should make \( i^{th} \) rule \( r_i \) in \( P \) and rule \( r_i' \) in \( P' \) produce a different rule-level decision. This requires that \( rb_i \) should evaluate to either *true* so that the \( i^{th} \) rule decision will be *permit* in \( P \) and *deny* in \( P' \) respectively or should *evaluate to an error* so that rule decision will be \( I(P) \) in \( P \) and \( I(D) \) in \( P' \). If \( rb_i \) evaluates to \( N/A \), the rule level decisions will be \( N/A \) in both \( P \) and \( P' \) in which case we cannot distinguish the faulty policy. Hence, formally, necessity constraint is \( rb_i \lor Error(rb_i) \).

iii) **Propagation constraint**

Given a faulty element produces different intermediate results, the *propagation constraint* must make \( P \) and \( P' \) produce a different policy-level decision. In other words, the different intermediate result from necessity constraint should contribute to producing a different policy-level decision. For this, any *permit* rule \( r_j \) (\( j > i \)) after \( r_i \), \( r_j \) should not evaluate to a *permit* decision, otherwise, \( d(P, q) = d(P', q) = \text{Permit} \). The test must make \( rb_j \) evaluate to \( N/A \) or *error* for each *permit* rule \( r_j \) (\( j > i \)). Therefore, the propagation constraint can be formalized as \( \neg rb_j \lor Error(rb_j) \) for any rule \( r_j = <rb_j, \text{Permit}> \) (\( j > i \)). Let, \( \mathcal{B} = “\neg rb_j \lor Error(rb_j) \) for any rule \( r_j = <rb_j, \text{Permit}> \) (\( j > i \)”\). \( \mathcal{B} \) is sufficient for propagation if necessity constraint is - \( rb_i \) evaluates to *true*. However, when \( rb_i \) evaluates to an error, \( \mathcal{B} \) is not sufficient for propagation.

We can verify that *when \( rb_i \) evaluates to true*, \( \mathcal{B} \) is sufficient for propagation by showing that it holds for all possible evaluation of other rules (all the rules in the policy except the current rule under consideration) as shown in Table 3.3. Let, the \( i^{th} \) rule in the
correct policy $P$ has *permit* effect, then that of the faulty policy $P'$ will be *deny* effect. The result of the RCA depends on the result of evaluation of other rules including that of $i^{th}$ rule. Table 3.3 presents the possible cases for the evaluation of other rules (in the second column), along with the evaluation of $i^{th}$ rule in $P$ and $P'$ (in third and fourth column respectively) and result of RCA for both $P$ and $P'$ (in fifth and sixth column respectively). As shown in Table 3.3, there is the possibility of four possible evaluation of other rules.

The first possible evaluation is that - all other rules evaluate to N/A and produces no effect. In this case, the effect of $i^{th}$ rule in $P$ - which is *permit* - will be the RCA level decision for $P$ and that of $P'$ will be *deny* since its $i^{th}$ rule has *deny* decision. When only one rule evaluates to *permit*, the result of *permit* overrides RCA is *permit*. Similarly, when only one rule evaluates to *deny*, the result of *permit*-overrides RCA is *deny*. As a result, the RCA level decision is *permit* and *deny* in $P$ and $P'$ respectively as shown in the fifth and the sixth column of the first row respectively. The policy level decision depends on the result of RCA and how policy target evaluates as discussed in Section 2.2.2 and Section 2.2.3. When $PT$ evaluates to *true* the policy-level result of $P$ and $P'$ will be the result of corresponding RCA i.e *permit* in $P$ and deny in $P'$. Similarly, when $PT$ evaluates to an error, the result of $P$ and $P'$ will be the Indeterminate of the result of RCA i.e the policy-level result of $P$ is $I(P)$ and that of $P'$ is $I(D)$. 
### Table 3.3: Possible Evaluations Of Other Rules when $R_i$ Evaluates to True

<table>
<thead>
<tr>
<th>other rules than $i^{th}$ rule</th>
<th>$i^{th}$ rule in P</th>
<th>$i^{th}$ rule in P'</th>
<th>RCA in P</th>
<th>RCA in P'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produces effect</td>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
<td>Deny</td>
</tr>
<tr>
<td></td>
<td>all other rules evaluate to N/A and produces no effect.</td>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
</tr>
<tr>
<td></td>
<td>One or more rule produces deny or I(D) effect and rest are N/A</td>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
</tr>
<tr>
<td></td>
<td>One or more rule produces I(P) effect and rest are N/A</td>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
</tr>
<tr>
<td></td>
<td>One or more rule produces I(P), one or more rule produces deny or I(D) effect and rest are N/A</td>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
</tr>
</tbody>
</table>

*Note: There is the possibility of another permit rule to be true but when it happens we could not detect the fault and the mutant will be equivalent to policy for such set of requests which makes another permit rule evaluates to true. Hence, for simplicity, we do not consider such cases in this table as well as in another table like this.*

The second possible evaluation is when one or more deny rule produces a deny or I(D) effect and the rest are N/A. In this case, since the RCA is *permit-overrides*, the result of RCA in $P$ is *permit* because $i^{th}$ rule has *permit* effect and if any rule evaluates to permit, the result of *permit-overrides* is permit. In $P'$, $i^{th}$ rule evaluated to deny, one or more rules evaluated to deny or I(D) and other rules evaluated to N/A i.e none of the rules evaluated to *permit* or I(P). When RCA is *permit-overrides*, if none of the rules evaluate to *permit* or I(P) and any one of the rules evaluate to *deny*, the RCA level decision will be *deny*. As a
result, the RCA level decision of $P'$ is deny. When $PT$ evaluates to true the result of $P'$ will be deny and that of $P$ is permit. Similarly, when $PT$ evaluates to an error the result of $P'$ is $I(D)$ and that of $P$ is $I(P)$ which are different.

The third possible evaluation is when one or more permit rules produces $I(P)$ effect and the rest are N/A. In this case, the result of RCA in $P$ is permit since $i^{th}$ rule has permit effect. For $P'$, $i^{th}$ rule evaluates to deny and since one or more rules evaluated to $I(P)$, the result of RCA will be $I(DP)$. As a result, when $PT$ evaluates to true the result of $P'$ will be $I(DP)$ and that of $P$ is permit which is different. Similarly, when $PT$ evaluates to an error the result of $P'$ is $I(DP)$ and that of $P$ is $I(P)$ which are different.

The fourth possible evaluation is one or more rules produces $I(P)$ effect, one or more rules produces deny or $I(D)$ effect and the rest are N/A. In this case, the result of RCA in $P$ is still permit since $i^{th}$ rule has permit effect and that of $P'$ will be $I(DP)$. As a result, when $PT$ evaluates to true the result of $P'$ will be $I(DP)$ and that of $P$ is permit which is different. Similarly, when $PT$ evaluates to an error the result of $P'$ is $I(DP)$ and that of $P$ is $I(P)$ which are different.

Hence, when $rb_i$ evaluates to true, $\phi$ is sufficient for propagation as it holds for all possible evaluation of other rules.

When $rb_i$ evaluates to an error, $\phi$ is not sufficient for propagation. Table 3.4 presents the rationale why $\phi$ is not sufficient for propagation when $rb_i$ evaluates to an error.

With similar reasoning as for Table 3.3, we can evaluate the result for RCA in $P$ and $P'$ as shown in Table 3.4. As listed in Table 3.4, the result of RCA in $P$ and $P'$ for the first three possible cases are different. Hence for them, $\phi$ is sufficient for propagation i.e fault
detection. However, for the fourth case, the result of RCA in both \( P \) and \( P' \) is \( I(DP) \). Since the result of RCA is not different and hence the policy level decision will not differ.

<table>
<thead>
<tr>
<th>Produces effect</th>
<th>other rules than ( i^{th} ) rule</th>
<th>( i^{th} ) rule in ( P )</th>
<th>( i^{th} ) rule in ( P' )</th>
<th>RCA in ( P )</th>
<th>RCA in ( P' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>No effect from other rules as all of them evaluates to N/A</td>
<td>I(P)</td>
<td>I(D)</td>
<td>I(P)</td>
<td>I(D)</td>
<td></td>
</tr>
<tr>
<td>One or more rule produces deny or I(D) effect and rest are N/A</td>
<td>I(P)</td>
<td>I(D)</td>
<td>Permit /I(P)</td>
<td>I(DP)</td>
<td></td>
</tr>
<tr>
<td>One or more rule produces I(P) effect and rest are N/A</td>
<td>I(P)</td>
<td>I(D)</td>
<td>I(DP)</td>
<td>I(D)</td>
<td></td>
</tr>
<tr>
<td>One or more rule produces I(P), one or more rule produces deny or I(D) effect and rest are N/A</td>
<td>I(P)</td>
<td>I(D)</td>
<td>I(DP)</td>
<td>I(DP)</td>
<td></td>
</tr>
</tbody>
</table>

Hence, \( \Phi \) is insufficient for propagation for the fourth case when \( rb_i \) evaluates to an error. As a result, when \( rb_i \) evaluates to an error, in addition to \( \Phi \), we need additional constraint in propagation constraint. The required additional constraint is - there should not exist a pair of the rules (excluding \( i^{th} \) rule) such that one of them has permit effect which evaluates to an error and other has deny effect which evaluates to true or error. If this constraint is satisfied, then the fourth case in Table 3.4 will never occur and we can distinguish the fault and if this constraint is violated the fourth case will occur in which case we cannot distinguish the faulty policy.
Hence, the propagation constraint when \( rb_i \) evaluates to an error is that - any permit rule \( r_j \) (\( j > i \)) after \( r_i \) should not evaluate to a permit decision and there should not exist a pair of the rules (excluding \( i^{th} \) rule) such that one of them has permit effect which evaluates to an error and other has deny effect which evaluates to true or error. Formally, additional constraint to \( \Phi \) can be specified as 
\[
\neg \left( \exists (p, d) \text{ such that } (i \neq p \neq d) \land (r_p = < rb_p, \text{permit}> \land \neg rb_j \lor \text{Error}(rb_j) \land (\text{Error}(rb_d) \lor rb_d)) \right).
\]

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is permit-overrides.

\[
(\{PT \lor \text{Error}(PT)\} \land \{rb_i\} \land \{\neg rb_j \lor \text{Error}(rb_j)\} \text{ for any permit rule } r_j = < rb_j, \text{permit}> \text{ for } j \neq i)
\]

\lor

\[
(\{PT \lor \text{Error}(PT)\} \land \{\text{Error}(rb_i)\} \land \{\neg rb_j \lor \text{Error}(rb_j)\} \text{ for any permit rule } r_j = < rb_j, \text{permit}> \text{ for } j \neq i) \land \neg \left( \exists (p, d) \text{ such that } (i \neq p \neq d) \land (r_p = < rb_p, \text{permit}> \land (r_d = < rb_d, \text{deny}> \lor \text{Error}(rb_d) \lor rb_d)\right)\}
\]

b) **Deny-overrides:**

i) **Reachability constraint**

The policy target should be true or should evaluate to an error. Further, when the rule-combining algorithm is deny-overrides, rule \( r_i \) will not be triggered if there is a deny rule before rule \( r_i \) that evaluates to a deny decision. Thus, the reachability constraint is that for any deny rule \( r_j \) (\( j < i \)) before rule \( r_i \), \( rb_j \) should not evaluate to true (i.e. it should be N/A or evaluates to an error).
Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with deny effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be \( \{PT \lor \text{Error}(PT)\} \land \{\neg rb_j \lor \text{Error}(rb_j)\} \text{ for any deny rule } r_j = <rb_j, \text{deny}> \text{ for } j < i\}

ii) **Necessity constraint**

\( rb_i \) should evaluate to either *true* so that the \( i^{th} \) rule decision will be *permit* in \( P \) and *deny* in \( P' \) respectively or should *evaluate to an error* so that rule decision will be \( I(P) \) in \( P \) and \( I(D) \) in \( P' \). If \( rb_i \) evaluates to *N/A*, the rule level decisions will be *N/A* in both \( P \) and \( P' \) in which case we can not distinguish the faulty policy. Hence, formally, necessity constraint is \( rb_i \lor \text{Error}(rb_i) \).

iii) **Propagation constraint**

Any *deny* rule \( r_j \) \((j > i)\) after \( r_i \), \( r_j \) should not evaluate to a *permit* decision, otherwise, \( d(P, q) = d(P', q) = \text{Deny} \). The test must make \( rb_j \) evaluate to *N/A* or *error* for each *deny* rule \( r_j \) \((j > i)\). Therefore, the propagation constraint can be formalized as \( \neg rb_j \lor \text{Error}(rb_j) \) for any rule \( r_j = <rb_j, \text{Deny}> \) \((j > i)\).

The afore-mentioned propagation constraint is sufficient for propagation if necessity constraint is \( rb_i \) evaluates to *true* but is not sufficient when \( rb_i \) evaluates to an error. With similar reasoning as for permit-overrides, we can state that when \( rb_i \) evaluates to an error, we need additional constraint for propagation. The required additional constraint is “any *deny* rule \( r_j \) \((j > i)\) after \( r_i \) should not evaluate to a *deny* decision and there should not exist a pair of the rules (excluding \( i^{th} \) rule) such that one of them has *deny* effect which evaluates to an error and other has *permit* effect which evaluates to *true* or *error*”. Formally,
additional constraint is \( \neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_p = \langle rb_p, \text{permit} \rangle) \land (r_d = \langle rb_d, \text{deny} \rangle) \land Error(rb_d) \land (Error(rb_p) \lor rb_p)) \).

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is \textit{deny-overrides}.

\[
(\{PT \lor Error(PT)\} \land \{rb\} \land \neg rb \lor Error(rb) \text{ for any deny rule } r_j = \langle rb_j, \text{deny} \rangle \text{ for } j \neq i)
\]
\[
\lor
\]
\[
(\{PT \lor Error(PT)\} \land \{Error(rb_d)\} \land \neg rb \lor Error(rb) \text{ for any permit rule } r_j = \langle rb_j, \text{deny} \rangle \text{ for } j \neq i) \land \neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = \langle rb_d, \text{deny} \rangle) \land (r_p = \langle rb_p, \text{permit} \rangle) \land Error(rb_d) \land (Error(rb_p) \lor rb_p))
\]

Note: The fault detection condition for \textit{permit-overrides} is symmetrical with fault detection condition for \textit{deny-overrides} such that the role of the permit and deny effect are interchanged. Hence, for simplicity, we would only consider \textit{permit-overrides} onwards for other faults.

c) \textit{deny-unless-permit:}

i) \textit{Reachability constraint}

The policy target should be \textit{true or} should evaluate to an error. When the RCA is \textit{deny-unless-permit}, rule \( r_i \) will not be triggered if there is a \textit{permit} rule before rule \( r_i \) that evaluates to a \textit{permit} decision. Thus, the reachability constraint is that for any \textit{permit} rule \( r_j = \langle rb_j, \text{permit} \rangle \) \((j < i)\) before rule \( r_i \), \( rb_j \) should not evaluate to \textit{true} (i.e it should be \textit{N/A} or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to \textit{true} or \textit{error} and all the previous rules with permit effect before the current rule under consideration should evaluate to \textit{N/A} or \textit{error} (i.e should not evaluate to \textit{true}).
In formal notation, it will be \( \{ PT \lor Error(PT) \} \land \{ \neg rb_j \lor Error(rb_j) \} \) for any permit rule \( r_j = <rb_j, permit> \) for \( j < i \).

ii) **Necessity constraint**

deny-unless-permit results permit decision only if at least one of the rules with permit effect evaluates to true, otherwise, it is deny in all other cases. As a result, we could not consider the error condition in the \( i^{th} \) rule because if there is error in the \( i^{th} \) rule target and/or condition, it will contribute to deny decision for the RCA in both P and P’ irrespective of its effect. Hence, both P and P’ will behave similarly if the current rule under consideration evaluates to an error. As a result, the necessity constraint should be the only \( rb_i \) evaluates to true. Formally, necessity constraint is \( rb_i \).

iii) **Propagation constraint**

Any permit rule \( r_j (j>i) \) after \( r_i \), \( r_j \) should not evaluate to a permit decision, otherwise, \( d(P, q) = d(P', q) = Permit \). The test must make \( rb_j \) evaluate to N/A or error for each permit rule \( r_j (j>i) \). Therefore, the propagation constraint can be formalized as \( \neg rb_j \lor Error(rb_j) \) for any rule \( r_j = <rb_j, Permit> \) (j>i).

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is deny-unless-permit.

\( \{ PT \lor Error(PT) \} \land \{ rb_i \} \land \{ \neg rb_j \lor Error(rb_j) \} \) for any permit rule \( r_j = <rb_j, Permit> \) (j!=i)

**d) permit-unless-deny:**

i) **Reachability constraint**

When the RCA is permit-unless-deny, rule \( r_i \) will not be triggered if there is a deny rule before rule \( r_i \) that evaluates to a permit decision. Thus, the reachability constraint is
that for any *deny* rule $r_j (j<i)$ before rule $r_i$, $rb_j$ should not evaluate to *true* (i.e. it should be N/A or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with deny effect before the current rule under consideration should evaluate to N/A or *error* (i.e. should not evaluate to *true*).

In formal notation, it will be $\{PT \lor Error(PT)\} \land \{\neg rb_j \lor Error(rb_j)\}$ for any deny rule $r_j = <rb_j, deny>$ for $j < i$.

**ii) Necessity constraint**

*permit-unless-deny* results *deny* only if at least one of the rules with *deny* effect evaluates to *true*, otherwise, it is *permit* in all other cases. As a result, we could not consider the *error* condition in the $i^{th}$ rule because if there is *error* in the $i^{th}$ rule target and/or condition, it will contribute to *permit* decision for the RCA in both P and P' irrespective of its effect. As a result, the necessity constraint should be the only $rb_i$ evaluates to *true*. Hence, formally, necessity constraint is $rb_i$.

**iii) Propagation constraint**

Any *deny* rule $r_j (j>i)$ after $r_i$, $r_j$ should not evaluate to a *deny* decision, otherwise, $d(P, q) = d(P', q) = Deny$. The test must make $rb_j$ evaluate to N/A or *error* for each deny rule $r_j (j>i)$. Therefore, the propagation constraint can be formalized as $\neg rb_j \lor Error(rb_j)$ for any rule $r_j = <rb_j, Deny> (j>i)$.

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is *permit-unless-deny*.

$\{\{PT \lor Error(PT)\} \land \{rb_i\} \land \{\neg rb_j \lor Error(rb_j)\}\} \text{ for any permit rule } r_j = <rb_j, Deny> \text{ for } j \neq i)$
Note: The fault detection condition for deny-unless-permit is symmetrical with fault detection condition for permit-unless-deny such that the role of the permit and deny effect are interchanged. Hence, for simplicity, we would only consider deny-unless-permit for other faults.

e) first-applicable:

i) Reachability constraint

When the rule-combining algorithm is first-applicable, rule \( r_i \) will not be triggered if there is any rule before rule \( r_i \) that evaluates to true or error. Thus, the reachability constraint is that - the policy target is true or evaluates to an error and for any rule \( r_j (j < i) \) before rule \( r_i \), \( rb_j \) should evaluate to N/A.

In formal notation, it will be \{PT \lor Error(PT)\} \land \{\neg rb_j\} for any rule \( r_j \) for \( j < i \)

ii) Necessity constraint

\( rb_i \) should evaluate to either true so that the \( ith \) rule decision will be permit in \( P \) and deny in \( P' \) respectively or should evaluate to an error so that rule decision will be \( I(P) \) in \( P \) and \( I(D) \) in \( P' \). If \( rb_i \) evaluates to N/A, the rule level decisions will be N/A in both \( P \) and \( P' \) in which case we cannot distinguish the faulty policy. Hence, formally, necessity constraint is \( rb_i \lor Error(rb_i) \).

iii) Propagation constraint

First-applicable RCA for CRE does not require explicit propagation constraint because reachability and necessity constraint is enough for fault detection.

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is first-applicable.

\{PT \lor Error(PT)\} \land \{rb_i \lor Error(rb_i)\} \land \{\neg rb_j\} for any rule \( r_j \) for \( j < i \)
3.2 FDC for Rule Target True (RTT)

Rule target *true* is a mutation operator which alters the rule target such that it will always evaluate to *true*. One of the transformation rules to make target always evaluate to *true* is to make it empty so that it will always evaluate to *true*. Since it has a fault in the target of the rule, it is under the category incorrect rule target.

Consider an XACML policy $P = \langle PT, RCA, RL \rangle$ and $P' = \langle PT, RCA, RL' \rangle$ where $RL = \langle r_1, ..., r_i, ..., r_n \rangle$, $RL' = \langle r_1, ..., r_i', ..., r_n \rangle$, $r_i = \langle rt_i, rc_i, re_i \rangle$ and $r_i' = \langle rt_i', rc_i, re_i \rangle$ such that $P'$ is similar to $P$ except the target $rt_i'$ of $i^{th}$ rule $r_i'$ in $P'$ always evaluates to *true*. Here, $P'$ is called the Rule Target True (RTT) Mutant of $P$. The fault detection condition for Rule Target True based on RCA are given below.

a) **Permit-overrides:**

i) **Reachability constraint**

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*). In formal notation, it will be $\{ PT \lor Error(PT) \} \land \{ \neg rb_j \lor Error(rb_j) \} \text{ for any permit rule } r_j = \langle rb_j, permit \rangle \text{ for } j < i$.

ii) **Necessity constraint**

The necessity constraint is that – rule target evaluates to *N/A* or *error* and rule condition evaluates to *true*. Formally, $(\neg rt_i \lor Error(rt_i)) \land rc_i$

iii) **Propagation constraint**

If policy target evaluates to *true*, $i^{th}$ rule target evaluates to *N/A* and $i^{th}$ rule effect is *deny*, then all other rules should evaluate to *N/A* or *error*. Further, there should not exist a
pair of the rules (excluding the current rule under consideration) such that one of them is
permit rule which evaluates to an error and other is deny rule which evaluates to true or
error.

\[\{\text{PT}\} \land \neg r_i \land r_i = <rb_i, \text{deny}> \land \{\neg r_j \lor \text{Error}(rb_j)\text{ for any rule } r_j \text{ for } j \neq i\} \land \{\neg(\exists(p,d)\text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, \text{deny}>) \land (r_p = <rb_p, \text{permit})\} \land \text{Error}(rb_p) \land (\text{Error}(rb_d) \lor rb_d))\}\]

If policy target evaluates to an error, \(i\)th rule target evaluates to N/A and \(i\)th rule effect
is deny, then all other rules with deny effect should evaluate to N/A and all other rules with
permit effect should not evaluate to true. Further, there should not exist a pair of the rules
(excluding the current rule under consideration) such that one of them is permit rule which
evaluates to an error and other is deny rule which evaluates to true or error.

\[\{\text{Error(PT)}\} \land \neg r_i \land r_i = <rb_i, \text{deny}> \land \{\neg r_j \lor \text{Error}(rb_j)\text{ for any permit rule } r_j = <rb_i, \text{permit}\text{ for } j \neq i\} \land \{\neg(\exists(p,d)\text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, \text{deny}>) \land (r_p = <rb_p, \text{permit})\} \land \text{Error}(rb_p) \land (\text{Error}(rb_d) \lor rb_d))\}\]

If policy target evaluates to true, \(i\)th rule target evaluates to N/A and \(i\)th rule effect is
permit, then all other permit rules should evaluate to N/A or error.

\[\{\text{PT}\} \land \neg r_i \land r_i = <rb_i, \text{permit}> \land \{\neg r_j \lor \text{Error}(rb_j)\text{ for any permit rule } r_j = <rb_i, \text{permit}\text{ for } j \neq i\}\]

If policy target evaluates to an error, \(i\)th rule target evaluates to N/A and \(i\)th rule effect
is permit, then all other rules with permit effect should evaluate to N/A if all deny rules
evaluate to N/A. However, if any deny rule evaluates to true or error, then another permit
effect can evaluate to N/A or error.
(\{\text{Error}(\text{PT})\} \land \neg r_i \land r_i = <rb_i, \text{permit}> \land \{\neg r_j \text{ for any rule } r_j \text{ for } j \neq i\} \lor \{\exists d (rb_d \lor \text{Error}(rb_d)) \land (r_d = <rb_d, \text{deny}> \land (\neg r_j \lor \text{Error}(rb_j)) \text{ for any } \text{deny rule } r_j = <rb_j, \text{deny}> \text{ for } j \neq i\})

If policy target evaluates to \text{true}, \(i^{th}\) rule target evaluates to an error and \(i^{th}\) rule effect is \text{deny}, then all other rules with \text{permit} effect should evaluate to \text{N/A} and all other rules should with \text{deny} effect should evaluate to \text{N/A} or \text{error}.

(\{\text{PT}\} \land \text{Error}(rt_i) \land r_i = <rb_i, \text{deny}> \land \{\neg r_j \lor \text{Error}(rb_j) \text{ for any } \text{deny rule } r_j = <rb_j, \text{deny}> \text{ for } j \neq i\})

If policy target evaluates to an error, \(i^{th}\) rule target evaluates to an error and \(i^{th}\) rule effect is \text{deny}, then we cannot detect the fault and mutant will behave the same as original policy for these set of requests.

If policy target evaluates to \text{true}, \(i^{th}\) rule target evaluates to an error and \(i^{th}\) rule effect is \text{permit}, then all other rules with \text{permit} effect should evaluate to \text{N/A}.

(\{\text{PT}\} \land \text{Error}(rt_i) \land r_i = <rb_i, \text{permit}> \land \{\neg r_j \text{ for any } \text{permit rule } r_j = <rb_j, \text{permit}> \text{ for } j \neq i\})

If policy target evaluates to an error, \(i^{th}\) rule target evaluates to an error and \(i^{th}\) rule effect is \text{permit}, then all other rules with \text{permit} effect should evaluate to \text{N/A} if all other deny rules evaluate to an error. However, if any one deny rule evaluates to \text{true} or \text{error}, another permit rule can evaluate to an error or \text{N/A}.

(\{\text{Error}(\text{PT})\} \land \text{Error}(rt_i) \land r_i = <rb_i, \text{permit}> \land \{\neg r_j \text{ for any } \text{rule } r_j \text{ for } j \neq i\} \lor \{\exists d (rb_d \lor \text{Error}(rb_d)) \land (r_d = <rb_d, \text{deny}> \land (\neg r_j \lor \text{Error}(rb_j)) \text{ for any } \text{permit rule } r_j = <rb_j, \text{permit}> \text{ for } j \neq i\})
Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is permit-overrides:

\[
(\{PT\} \land \neg rt_i \land rc_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) for any rule r_j for j \neq i\} \land
\neg (\exists (p,d) such that (i \neq p \neq d) \land (r_d = <rb_d, deny>) \land (r_p = <rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d)))
\]

\[
\lor
\]

\[
(\{Error(PT)\} \land \neg rt_i \land rc_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) for any permit rule r_j = <rb_i, permit> for j \neq i\})
\]

\[
\lor
\]

\[
(\{Error(PT)\} \land \neg rt_i \land rc_i \land r_i = <rb_i, permit> \land \{\neg rb_j for any rule r_j for j \neq i\} \lor \{\exists d (rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j)) for any permit rule r_j = <rb_i, permit> for j \neq i\})
\]

\[
\lor
\]

\[
(\{PT\} \land Error(rt_i) \land rc_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) for any deny rule r_j = <rb_i, deny> for j \neq i\} \land \{rb_j, deny> for j \neq i\} \land \{\neg rb_j for any permit rule r_j = <rb_j, permit> for j \neq i\})
\]

\[
\lor
\]

\[
(\{PT\} \land Error(rt_i) \land rc_i \land r_i = <rb_i, permit> \land \{\neg rb_j for any permit rule r_j = <rb_j, permit> for j \neq i\})
\]
\[
(\{Error(PT)\} \land Error(rt) \land rc_i \land r_i = <rb_i, permit> \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j \neq i\} \lor
\{\exists d (rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j)) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \neq i\})
\]

We can verify the validity of this fault detection condition by adopting similar reasoning that we applied for CRE. Let \(i^{th}\) rule have \textit{deny} effect. The result of the RCA depends on the result of evaluation of other rules as well as that of \(i^{th}\) rule. Table 3.5 presents the possible cases for evaluation of other rules and result of RCA for both \(P\) and \(P'\).

With similar reasoning as for Table 3.3, we can list the records in Table 3.5. The final case in Table 3.5, when one or more rules (excluding \(i^{th}\) rule) produces \(I(P)\) effect and one or more other rules produces deny effect and rest of the rules are \(N/A\), the result of RCA is similar in both \(P'\) and \(P\) (which implies that we cannot detect fault under those conditions).

Additionally, in the second case - when one or more of the other rules evaluate to \textit{deny} and rest as \(N/A\), the result of RCA in \(P\) and \(P'\) are same. Hence from Table 3.5, we can conclude that if the effect of \(i^{th}\) rule is \textit{deny} then for fault detection, all the rules except the \(i^{th}\) rule should not be \textit{true} and there should not exists a pair of the rules (excluding \(i^{th}\) rule) such that one of them has \textit{deny} effect, another has \textit{permit} effect, \textit{permit} rule evaluates to an error, and \textit{deny} rule evaluates to \textit{true} or \textit{error}. 

Table 3.5: Possible Evaluations of Other Rules When \( R_t \) Evaluates to \( N/A \) and \( R_i \) Is Deny Rule in \( P \)

<table>
<thead>
<tr>
<th></th>
<th>other rules than ( i^{th} ) rule</th>
<th>( i^{th} ) rule in ( P )</th>
<th>( i^{th} ) rule in ( P' )</th>
<th>RCA in ( P )</th>
<th>RCA in ( P' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produces effect</td>
<td>No effect from other rules as all of them evaluates to ( N/A )</td>
<td>N/A</td>
<td>deny</td>
<td>N/A</td>
<td>deny</td>
</tr>
<tr>
<td></td>
<td>One or more rule produces ( deny ) effect and rest are ( N/A )</td>
<td>N/A</td>
<td>deny</td>
<td>deny</td>
<td>deny</td>
</tr>
<tr>
<td></td>
<td>One or more rule produces ( I(D) ) effect and rest are ( N/A )</td>
<td>N/A</td>
<td>deny</td>
<td>I(D)</td>
<td>deny</td>
</tr>
<tr>
<td></td>
<td>One or more rule produces ( I(P) ) effect and rest are ( N/A )</td>
<td>N/A</td>
<td>deny</td>
<td>I(P)</td>
<td>I(DP)</td>
</tr>
<tr>
<td></td>
<td>One or more rule produces ( I(P) ), one or more rule produces ( deny ) or ( I(D) ) effect and rest are ( N/A )</td>
<td>N/A</td>
<td>deny</td>
<td>I(DP)</td>
<td>I(DP)</td>
</tr>
</tbody>
</table>

Further, if \( PT \) evaluates to an error then the result of both \( P \) and \( P' \) will be \( I(D) \) for the third case when one or more other rules with \( deny \) effect evaluate to \( I(D) \). Hence, when \( PT \) evaluates to an error, the fault detection condition is such that other rules with \( permit \) effect can evaluate to an error while that with \( deny \) effect should evaluate to \( N/A \).

Now consider the effect of \( i^{th} \) rule is \( permit \) instead of \( deny \). In such case, Table 3.6 presents the possible cases for evaluation of other rules and result of RCA for both \( P \) and \( P' \).
It is evident from the cases in Table 3.6 that when the effect of the $i^{th}$ rule is **permit**, it can identify the fault if none of the other permit rule evaluates to **true**.

Further, if $PT$ evaluates to an error then result of both $P$ and $P'$ will be $I(P)$ for the fourth case when the rule with the permit effect evaluates to an error. Hence, when $PT$ evaluates to an error, the fault detection condition is such that rules with the permit effect can only evaluate to $N/A$. However, if there is an another deny rule which does not evaluate to $N/A$ then there could be a permit rule which evaluates to an error which is nothing but the final case in which case we can distinguish the fault.
Now, let’s consider the situation when $i^{th}$ rule target evaluates to an error. Table 3.7 presents the possible cases for evaluation of other rules and result of RCA for both $P$ and $P'$. From Table 3.7, it is evident that fault can be detected only when other rules with permit effect evaluate to N/A and rules with deny effect do not evaluate to true.

**Table 3.7: Possible Evaluations of Other Rules When $R_t$, Evaluates to Error And $R_i$ is Deny Rule in $P$**

<table>
<thead>
<tr>
<th>other rules than $i^{th}$ rule</th>
<th>$i^{th}$ rule in $P$</th>
<th>$i^{th}$ rule in $P'$</th>
<th>RCA in $P$</th>
<th>RCA in $P'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produces effect</td>
<td>No effect from other rules as all of them evaluates to N/A</td>
<td>I(D)</td>
<td>deny</td>
<td>I(D)</td>
</tr>
<tr>
<td>One or more rule produces deny effect and rest are N/A</td>
<td>I(D)</td>
<td>deny</td>
<td>deny</td>
<td>deny</td>
</tr>
<tr>
<td>One or more rule produces $I(D)$ effect and rest are N/A</td>
<td>I(D)</td>
<td>deny</td>
<td>I(D)</td>
<td>deny</td>
</tr>
<tr>
<td>One or more rule produces $I(P)$ effect and rest are N/A</td>
<td>I(D)</td>
<td>deny</td>
<td>I(DP)</td>
<td>I(DP)</td>
</tr>
<tr>
<td>One or more rule produces $I(P)$, one or more rule produces deny or $I(P)$ effect and rest are N/A</td>
<td>I(D)</td>
<td>deny</td>
<td>I(DP)</td>
<td>I(DP)</td>
</tr>
</tbody>
</table>

Further, if $PT$ evaluates to an error then result of both $P$ and $P'$ will be the same in all cases and hence mutant behaves similarly as original policy.
Now consider the effect of \( i^{th} \) rule is \textit{permit} instead of \textit{deny}. In such case, Table 3.8 presents the possible cases for evaluation of other rules and result of RCA for both P and P’.

\textbf{Table 3.8. Possible Evaluations of Other Rules when} \( R_{ti} \text{ Evaluates to Error and } R_i \text{ is Permit Rule in } P \)

<table>
<thead>
<tr>
<th>Produces effect</th>
<th>other rules than ( i^{th} ) rule</th>
<th>( i^{th} ) rule in P</th>
<th>( i^{th} ) rule in P’</th>
<th>RCA in P</th>
<th>RCA in P’</th>
</tr>
</thead>
<tbody>
<tr>
<td>No effect from other rules as all of them evaluates to N/A</td>
<td>I(P)</td>
<td>permit</td>
<td>I(P)</td>
<td>permit</td>
<td></td>
</tr>
<tr>
<td>One or more rule produces ( deny ) effect and rest are N/A</td>
<td>I(P)</td>
<td>permit</td>
<td>I(DP)</td>
<td>permit</td>
<td></td>
</tr>
<tr>
<td>One or more rule produces ( I(D) ) effect and rest are N/A</td>
<td>I(P)</td>
<td>permit</td>
<td>I(DP)</td>
<td>permit</td>
<td></td>
</tr>
<tr>
<td>One or more rule produces ( I(P) ) effect and rest are N/A</td>
<td>I(P)</td>
<td>permit</td>
<td>I(P)</td>
<td>permit</td>
<td></td>
</tr>
<tr>
<td>One or more rule produces ( I(P) ), one or more rule produces ( deny ) or ( I(D) ) effect and rest are N/A</td>
<td>I(P)</td>
<td>permit</td>
<td>I(DP)</td>
<td>permit</td>
<td></td>
</tr>
</tbody>
</table>

It is evident from the above cases in Table 3.8 that when the effect of the \( i^{th} \) rule is \textit{permit}, it can identify the fault if all other permit rules does not evaluate to \textit{true}.

Further, if \( PT \) evaluates to an error then the fault detection condition is such that other rules with \textit{permit} effect cannot evaluate to \textit{true} or \textit{error} if all of the \textit{deny} rules evaluate to
N/A. However, if at least one other deny rule evaluates to true or error, then another permit rule can evaluate to an error or N/A.

*Note: We can establish the validity of the fault detection condition for all faults with similar reasoning as we did above for CRE and RTT. Hence, further in this work for simplicity, we only specify the fault detection condition precisely.*

**b) deny-unless-permit:**

**i) Reachability constraint**

The policy target should be true or should evaluate to an error. For any permit rule \( r_j = <rb_j, permit> \) \( j < i \) before rule \( r_i \), \( rb_j \) should not evaluate to true (i.e it should be N/A or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to true or error and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A or error (i.e should not evaluate to true).

In formal notation, it will be \( \{PT \lor Error(PT)\} \land \{\neg rb_j \lor Error(rb_j)\} \) for any permit rule \( r_j = <rb_j, permit> \) for \( j < i \)

**ii) Necessity constraint**

The target of the rule under consideration should evaluate to N/A or error and condition should evaluate to true. Formally, \( (\neg rt_i \lor Error(rt_i)) \land rc_i \)

**iii) Propagation constraint**

If rule under consideration is deny rule, then the mutant is equivalent because the rule level decision in mutant will be deny, and, in original policy, it will be N/A or error. However, for deny-unless-permit RCA, anything other than permit effect results in deny decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both of them will behave the same.
Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect after the \( i^{th} \) rule should evaluate to N/A or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is *deny-unless-permit*.

\[
(\{PT \lor Error(PT)\} \land \{\neg rt_i \lor Error(rt_i)\} \land \neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_j, permit > \text{ for } j \neq i))
\]

c)  

**first-applicable:**

i)  

*Reachability constraint*

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A.

ii)  

*Necessity constraint*

If policy target evaluates to *true*, the necessity constraint is that – rule target evaluates to N/A or *error* and rule condition evaluates to *true*.

If policy target evaluates to an error, the necessity constraint is that – rule target evaluates to N/A and rule condition evaluates to *true*.

iii)  

*Propagation constraint*

If policy target evaluates to *true* and \( i^{th} \) rule target evaluates to *false*, then for all other rules which have the same effect as that of \( i^{th} \) rule should evaluate to N/A or *error*.

If policy target evaluates to an error and \( i^{th} \) rule target evaluates to *false*, then for all other rules which have the same effect as that of \( i^{th} \) rule should evaluate to N/A.

If policy target evaluates to *true* and \( i^{th} \) rule target evaluates to an error, then explicit propagation constraint is not required such that reachability and necessity constraint are enough for fault detection.
Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is first-applicable.

\((\{PT\} \land \neg rt_i \land rc_i \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \lor Error(rb_j) \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\})\)

\(\lor\)

\((Error(PT)) \land \neg rt_i \land rc_i \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\})\)

\(\lor\)

\((PT \land Error(rt_i) \land rc_i \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\})\)

3.3 FDC for Rule Target False (RTF)

Rule target false is a mutation operator which alters the rule target such that it will always evaluate to false. One of the transformation rules to make target always evaluate to false is to introduce new constraint in the target with a random attribute which will always be false. Since it has a fault in the target of the rule, it is under the category incorrect rule target.

Consider an XACML policy \(P = <PT, RCA, RL>\) and \(P' = <PT, RCA, RL'>\) where \(RL = \langle r_1, \ldots, r_i, \ldots, r_n \rangle\), \(RL' = \langle r_1, \ldots, r_i', \ldots, r_n \rangle\), \(r_i = \langle rt_i, rc_i, re_i \rangle\) and \(r_i' = \langle rt_i, rc_i, re_i \rangle\) such that \(P'\) is similar to \(P\) except the target \(rt_i'\) of \(i^{th}\) rule \(r_i\) in \(P'\) always evaluates to false. Here, \(P'\) is called the Rule Target False (RTF) Mutant of \(P\). The fault detection condition for Rule Target False based on RCA are given below.

a) Permit-overrides:

i) Reachability constraint
Policy target should evaluate to *true* or *error* and all the previous rule with permit effect before the current rule under consideration should evaluate to N/A or *error* (i.e. should not evaluate to *true*).

In formal notation, it will be \( \{PT \lor Error(PT)\} \land \{\neg rb_j \lor Error(rb_j)\} \) for any permit rule \( r_j = <rb_j, permit> \) for \( j < i \)

ii) **Necessity constraint**

The necessity constraint is that – rule target evaluates to *true* or *error* and rule condition evaluates to *true*. Formally, \( (rt_i \lor Error(rt_i)) \land rc_i \)

iii) **Propagation constraint**

If policy target evaluates to *true*, \( i^{th} \) rule target evaluates to *true* and \( i^{th} \) rule effect is *deny*, then all other rules should evaluate to N/A or *error*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

\( \{\{PT\}\} \land rt_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j)\} \) for any rule \( r_j \) for \( j \neq i \) \( \land \{\neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, deny>) \land (r_p = <rb_p, permit>) \land (Error(rb_p) \land (Error(rb_d) \lor rb_d)))\} \)

If policy target evaluates to an error, \( i^{th} \) rule target evaluates to *true* and \( i^{th} \) rule effect is *deny*, then all other rules with *deny* effect should evaluate to N/A and all other rules with *permit* effect should not evaluate to *true*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*. 
If policy target evaluates to true, \(i^{th}\) rule target evaluates to true and \(i^{th}\) rule effect is permit, then all other permit rule should evaluate to N/A or error.

\[
(\{\text{Error(PT)}\} \land r_{ti} \land r_i = <rb_i, \text{permit}> \land \{\neg rb_j \lor \text{Error(rb}_j)\} \text{ for any permit rule } r_j = <rb_i, \text{permit}> \text{ for } j \neq i) \land \{\neg (\exists p, d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, \text{deny}> \lor \text{Error(rb}_d) \lor \text{Error(rb}_d)) \text{ for any permit rule } r_j = <rb_i, \text{permit}> \text{ for } j \neq i}\}
\]

If policy target evaluates to an error, \(i^{th}\) rule target evaluates to true and \(i^{th}\) rule effect is permit, then all other rules with permit effect should evaluate to N/A if all deny rules evaluate to N/A. However, if any deny rule evaluates to true or error, then another permit effect can evaluate to N/A or error.

\[
(\{\text{Error(PT)}\} \land r_{ti} \land r_i = <rb_i, \text{permit}> \land \{\neg rb_j \lor \text{Error(rb}_j)\} \text{ for any permit rule } r_j = <rb_i, \text{permit}> \text{ for } j \neq i) \lor \{\exists d (rb_d \lor \text{Error(rb}_d)) \land (r_d = <rb_d, \text{deny}> \lor \neg rb_j \lor \text{Error(rb}_j)) \text{ for any permit rule } r_j = <rb_i, \text{permit}> \text{ for } j \neq i\}
\]

If policy target evaluates to true or error, \(i^{th}\) rule target evaluates to an error and \(i^{th}\) rule effect is deny, then all other rules with deny effect should evaluate to N/A and all other rules with permit effect should evaluate to N/A or error.

\[
(\{\text{PT} \lor \text{Error(PT)}\} \land \text{Error(rt}_i) \land r_i = <rb_i, \text{deny}> \land \{\neg rb_j \lor \text{Error(rb}_j)\} \text{ for any permit rule } r_j = <rb_i, \text{permit}> \text{ for } j \neq i) \land \{\neg rb_j \lor \text{Error(rb}_j) \text{ for any deny rule } r_j = <rb_j, \text{deny}> \text{ for } j \neq i\}
\]

If policy target evaluates to true or error, \(i^{th}\) rule target evaluates to an error and \(i^{th}\) rule effect is permit, then all other rules with permit effect should evaluate to N/A.
Combining all constraints as a single constraint, we get the following constraint for fault detection of RTF when RCA is permit-overrides:

\[
({\{PT \lor Error(PT)\}} \land Error(r_{t_i}) \land r_i = <rb_{i}, permit> \land \{\neg rb_{j} \forall \text{any permit rule } r_j = <rb_{j}, permit> \text{ for } j \neq i\})
\]

\[
\lor
\]

\[
({\{Error(PT)\}} \land r_{t_i} \land r_{c_i} \land r_i = <rb_{i}, deny> \land \{\neg rb_{j} \forall \text{ any permit rule } r_j = <rb_{j}, permit> \text{ for } j \neq i\} \land \{\neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_{d}, deny>) \land (r_p = <rb_{p}, permit>) \land Error(rb_{p}) \land (Error(rb_{d}) \lor rb_{d}))\})
\]

\[
\lor
\]

\[
({\{PT\}} \land r_{t_i} \land r_{c_i} \land r_i = <rb_{i}, deny> \land \{\neg rb_{j} \forall \text{ any deny rule } r_j = <rb_{j}, deny> \text{ for } j \neq i\} \land \{\neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_{d}, deny>) \land (r_p = <rb_{p}, permit>) \land Error(rb_{p}) \land (Error(rb_{d}) \lor rb_{d}))\})
\]

\[
\lor
\]

\[
({\{PT\}} \land r_{t_i} \land r_{c_i} \land r_i = <rb_{i}, permit> \land \{\neg rb_{j} \forall \text{ any permit rule } r_j = <rb_{j}, permit> \text{ for } j \neq i\})
\]

\[
\lor
\]

\[
({\{Error(PT)\}} \land r_{t_i} \land r_{c_i} \land r_i = <rb_{i}, permit> \land \{\neg rb_{j} \forall \text{ any rule } r_j = <rb_{j}, deny> \text{ for } j \neq i\} \lor \{\exists d (rb_{d} \lor Error(rb_{d})) \land (r_d = <rb_{d}, deny>) \land (\neg rb_{j} \lor Error(rb_{j})) \forall \text{ any permit rule } r_j = <rb_{j}, permit> \text{ for } j \neq i\})
\]

\[
\lor
\]

\[
({\{PT \lor Error(PT)\}} \land Error(r_{t_i}) \land r_{c_i} \land r_i = <rb_{i}, deny> \land \{\neg rb_{j} \forall \text{ any permit rule } r_j = <rb_{j}, permit> \text{ for } j \neq i\} \lor \{\neg rb_{j} \forall \text{ any deny rule } r_j = <rb_{j}, deny> \text{ for } j \neq i\}
\]

\[
\lor
\]

\[
({\{PT \lor Error(PT)\}} \land Error(r_{t_i}) \land r_{c_i} \land r_i = <rb_{i}, deny> \land \{\neg rb_{j} \forall \text{ any permit rule } r_j = <rb_{j}, permit> \text{ for } j \neq i\} \lor \{\neg rb_{j} \forall \text{ any deny rule } r_j = <rb_{j}, deny> \text{ for } j \neq i\}
\]

\[
\lor
\]
Let rule $r_i = \langle rb_i, \text{permit} \rangle$ and $\{PT \lor \text{Error}(PT)\} \land \text{Error}(rt) \land rc_i \land r_i = \langle rb_i, \text{permit} \rangle \land \{\neg rb_j \text{ for any permit rule } r_j = \langle rb_j, \text{permit} \rangle \text{ for } j \neq i\}$

b) **deny-unless-permit:**

i) **Reachability constraint**

The policy target should be true or should evaluate to an error. For any permit rule $r_j = \langle rb_j, \text{permit} \rangle$ (j<i) before rule $r_i$, $rb_j$ should not evaluate to true (i.e it should be N/A or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to true or error and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A or error (i.e should not evaluate to true).

In formal notation, it will be $\{PT \lor \text{Error}(PT)\} \land \{\neg rb_j \lor \text{Error}(rb_j) \text{ for any permit rule } r_j = \langle rb_j, \text{permit} \rangle \text{ for } j < i\}$

ii) **Necessity constraint**

The target of the rule under consideration should evaluate to true and condition should evaluate to true. Formally, $rt_i \land rc_i$.

iii) **Propagation constraint**

If rule under consideration is deny rule, then the mutant is equivalent because the rule level decision in mutant will be N/A, and, in original policy, it will be true or error. However, for deny-unless-permit RCA, anything other than permit effect results in deny decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both will behave the same.

Hence, the rule under consideration should be permit rule such that all the rules with permit effect evaluate to N/A or error.
Combining all constraints as a single constraint, we get the following constraint for fault detection of RTF when RCA is \textit{deny-unless-permit}.

\[(\{PT \lor Error(PT)\} \land \{(rt_i) \land rc_i\} \land \{\neg rb_j \lor Error(rb_i)\text{ for any permit rule } r_j = \text{\textless rb_i permit for } j \neq i})\] 

c) \textit{first-applicable:}

i) \textit{Reachability constraint}

Policy target should evaluate to \textit{true} or \textit{error} and all the previous rules with permit effect before the current rule under consideration should evaluate to \textit{N/A}.

ii) \textit{Necessity constraint}

If policy target evaluates to \textit{true}, the necessity constraint is that – rule target evaluates to \textit{N/A} or \textit{error} and rule condition evaluates to \textit{true}.

If policy target evaluates to an error, the necessity constraint is that – rule target evaluates to \textit{N/A} and rule condition evaluates to \textit{true}.

iii) \textit{Propagation constraint}

If policy target evaluates to \textit{true} and \textit{i}th rule target evaluates to \textit{true}, then for all other rules which have the same effect as that of \textit{i}th rule should evaluate to \textit{N/A} or \textit{error}.

If policy target evaluates to \textit{true} and \textit{i}th rule target evaluates to an error, then for all other rules after the \textit{i}th rule which has the same effect as that of \textit{i}th rule should evaluate to \textit{N/A} or \textit{true}.

If policy target evaluates to an error and \textit{i}th rule target evaluates to an error or \textit{true}, then for all other rules after the \textit{i}th rule which has the same effect as that of \textit{i}th rule should evaluate to \textit{N/A}.
Combining all constraints as a single constraint, we get the following constraint for fault detection of RTF when RCA is first-applicable.

\[(\{PT\} \land rt \land rc \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \lor Error(rb_j) \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\}) \lor \]

\[(\{PT\} \land Error(rt) \land rc \land \{\neg rb_j \lor rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\}) \lor \]

\[(\{Error(PT)\} \land (rt \lor Error(rt)) \land rc \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\}) \]

3.4 FDC for Rule Condition True (RCT)

Rule condition \textit{true} is a mutation operator which alters the rule condition such that it will always evaluate to \textit{true}. One of the transformation rules to make condition always evaluate to \textit{true} is to make it empty so that it will always evaluate to \textit{true}. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy \(P = \langle PT, RCA, RL \rangle\) and \(P' = \langle PT, RCA, RL' \rangle\) where \(RL = \langle r_i, ..., r_i, ..., r_i \rangle, RL' = \langle r_i, ..., r_i, ..., r_i \rangle, r_i = \langle rt_i, rc_i, re_i \rangle\) and \(r_i' = \langle rt_i', rc_i, re_i \rangle\) such that \(P'\) is similar to \(P\) except the condition \(rc_i'\) of \(i^{th}\) rule \(r_i'\) in \(P'\) always evaluates to \textit{true}. Here, \(P'\) is called the Rule Condition True (RCT) Mutant of \(P\). The fault detection condition for Rule Condition True based on RCA are given below.

a) Permit-overrides:

i) \textit{Reachability constraint}
Policy target should evaluate to \textit{true} or \textit{error} and all the previous rules with permit effect before the current rule under consideration should evaluate to \textit{N/A} or \textit{error} (i.e. should not evaluate to \textit{true}).

In formal notation, it will be \( \{PT \lor Error(PT)\} \land \{\neg rb_j \lor Error(rb_j)\} \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j < i\)

\(\text{ii) Necessity constraint}\)

The necessity constraint is that – rule condition evaluates to \textit{N/A} or \textit{error} and rule target evaluates to \textit{true}.

Formally, \( (\neg rc_i \lor Error(rc_i)) \land rt_i \)

\(\text{iii) Propagation constraint}\)

If policy target evaluates to \textit{true}, \(i^{th}\) rule condition evaluates to \textit{N/A} and \(i^{th}\) rule effect is \textit{deny}, then all other rules should evaluate to \textit{N/A} or \textit{error}. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is \textit{permit} rule which evaluates to an error and other is \textit{deny} rule which evaluates to \textit{true} or \textit{error}.

\( (\{PT\} \land \neg rc_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j)\} \text{ for any rule } r_j \text{ for } j \neq i\} \land \{\neg (\exists(p,d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, deny>) \land (r_p = <rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d)))\})\)

If policy target evaluates to an error, \(i^{th}\) rule condition evaluates to \textit{N/A} and \(i^{th}\) rule effect is \textit{deny}, then all other rules with \textit{deny} effect should evaluate to \textit{N/A} and all other rules with \textit{permit} effect should not evaluate to \textit{true}. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is \textit{permit} rule which evaluates to an error and other is \textit{deny} rule which evaluates to \textit{true} or \textit{error}. 
\((\{\text{Error}(PT)\} \land \neg r_{ci} \land r_i = <rb_i, \text{deny}> \land \{\neg rb_j \lor \text{Error}(rb_j) \text{ for any permit rule } r_j = <rb_j, \text{permit}> \} \land \{\neg rb_j \text{ for any deny rule } r_j = <rb_j, \text{deny}> \text{ for } j \neq i\}) \land \{\neg (\exists (p, d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, \text{deny}>)) \land (r_p = <rb_p, \text{permit}>)) \land \text{Error}(rb_p) \land (\text{Error}(rb_d) \lor rb_d))\})\)

If policy target evaluates to \textit{true}, \textit{i}th rule condition evaluates to \textit{N/A} and \textit{i}th rule effect is \textit{permit}, then all other permit rules should evaluate to \textit{N/A} or \textit{error}.

\((\{PT\} \land \neg r_{ci} \land r_i = <rb_i, \text{permit}> \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j \neq i\} \land \{\neg (\exists (p, d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, \text{deny}>)) \land (r_p = <rb_p, \text{permit}>)) \land \text{Error}(rb_p) \land (\text{Error}(rb_d) \lor rb_d))\})\)

If policy target evaluates to \textit{true}, \textit{i}th rule condition evaluates to \textit{N/A} and \textit{i}th rule effect is \textit{permit}, then all other rules with \textit{permit} effect should evaluate to \textit{N/A} if all \textit{deny} rules evaluate to \textit{N/A}. However, if any \textit{deny} rule evaluates to \textit{true} or \textit{error}, then another permit effect can evaluate to \textit{N/A} or \textit{error}.

\((\{\text{Error}(PT)\} \land \neg r_{ci} \land r_i = <rb_i, \text{permit}> \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j \neq i\} \land \{\neg (\exists (p, d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, \text{deny}>)) \land (r_p = <rb_p, \text{permit}>)) \land \text{Error}(rb_p) \land (\text{Error}(rb_d) \lor rb_d))\})\)

If policy target evaluates to \textit{true}, \textit{i}th rule condition evaluates to an error and \textit{i}th rule effect is \textit{deny}, then all other rules with \textit{permit} effect should evaluate to \textit{N/A} and all other rules should with \textit{deny} effect should evaluate to \textit{N/A} or \textit{error}.

\((\{PT\} \land \text{Error}(rc_i) \land r_i = <rb_i, \text{deny}> \land \{\neg rb_j \lor \text{Error}(rb_j) \text{ for any deny rule } r_j = <rb_j, \text{deny}> \} \land \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, \text{permit}> \text{ for } j \neq i\})\)

If policy target evaluates to an error, \textit{i}th rule target evaluates to an error and \textit{i}th rule effect is \textit{deny}, then we cannot detect the fault and mutant will behave the same as original policy for these set of requests.
If policy target evaluates to true, \(i\)th rule condition evaluates to an error and \(i\)th rule effect is permit, then all other rules with permit effect should evaluate to N/A.

\[
\{PT\} \land \text{Error}(r_{ci}) \land r_i = <rb_i, \text{permit}> \land \{\neg r_j \text{for any rule } r_j = <rb_j, \text{permit}> \text{ for } j \neq i}\)
\]

If policy target evaluates to an error, \(i\)th rule condition evaluates to an error and \(i\)th rule effect is permit, then all other rules with permit effect should evaluate to N/A if all other deny rules evaluate to an error. However, if any one deny rule evaluates to true or error, another permit rule can evaluate to an error or N/A.

\[
\{\text{Error}(PT)\} \land \text{Error}(r_{ci}) \land r_i = <rb_i, \text{permit}> \land \{\neg r_j \text{for any rule } r_j \text{ for } j \neq i\} \lor \{\exists d \ (r_d \lor \text{Error}(rb_d)) \land (r_d = <rb_d, \text{deny}>) \land (\neg r_j \lor \text{Error}(rb_j)) \text{ for any permit rule } r_j = <rb_j, \text{permit}> \text{ for } j \neq i\}
\]

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCT when RCA is permit-overrides:

\[
\{PT\} \land \neg r_{ci} \land r_{ti} \land r_i = <rb_i, \text{deny}> \land \{\neg r_j \lor \text{Error}(rb_j) \text{ for any rule } r_j \text{ for } j \neq i\} \land \\
\neg (\exists (p,d) \text{ such that } i \neq p \neq d) \land (r_d = <rb_d, \text{deny}>) \land (r_p = <rb_p, \text{permit}>) \land \text{Error}(rb_p) \land (\text{Error}(rb_d) \lor rb_d))
\]

\[
\lor
\]

\[
\{\text{Error}(PT)\} \land \neg r_{ci} \land r_{ti} \land r_i = <rb_i, \text{deny}> \land \{\neg r_j \lor \text{Error}(rb_j) \text{ for any permit rule } r_j = <rb_i, \text{permit}> \text{ for } j \neq i\} \land \neg (\exists (p,d) \text{ such that } i \neq p \neq d) \land (r_d = <rb_d, \text{deny}>) \land (r_p = <rb_p, \text{permit}>) \land \text{Error}(rb_p) \land (\text{Error}(rb_d) \lor rb_d))
\]

\[
\lor
\]
(\{PT\} \land \neg rc_i \land rt_i \land r_i = <rb_i, permit> \land \{\neg rb_j \lor Error(rb_j)\} for any permit rule r_j = <rb_i, permit> for j \neq i))

\lor

(\{Error(PT)\} \land \neg rc_i \land rt_i \land r_i = <rb_i, permit> \land \{\neg rb_j for any rule r_j for j \neq i\} \lor \exists d ( rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j)) for any permit rule r_j = <rb_i, permit> for j \neq i )])

\lor

(\{PT\} \land Error(rc_i) \land rt_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j)\} for any deny rule r_j = <rb_i, deny> for j \neq i \} \land \{ \neg rb_j for any permit rule r_j = <rb_i, permit> for j \neq i \})

\lor

(\{PT\} \land Error(rc_i) \land rt_i \land r_i = <rb_i, permit> \land \{\neg rb_j for any permit rule r_j = <rb_i, permit> for j \neq i \})

\lor

(\{Error(PT)\} \land Error(rc_i) \land rt_i \land r_i = <rb_i, permit> \land \{\neg rb_j for any rule r_j for j \neq i\} \lor \exists d ( rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j)) for any permit rule r_j = <rb_i, permit> for j \neq i )])

\textbf{b) deny-unless-permit:}

\textit{i) Reachability constraint}

The policy target should be \textit{true} or should evaluate to an error. For any permit rule $r_j = <rb_j, permit> (j<i)$ before rule $r_i$, $rb_j$ should not evaluate to \textit{true} (i.e. it should be N/A or evaluates to an error).
Hence, the reachability constraint is - policy target evaluates to true or error and all the previous rule with permit effect before the current rule under consideration should evaluate to N/A or error (i.e should not evaluate to true).

In formal notation, it will be \( \{PT \lor \text{Error}(PT)\} \land \{\neg rb_j \lor \text{Error}(rb_j)\} \) for any permit rule \( r_j = <rb_j, \text{permit}> \) for \( j < i \)

ii) **Necessity constraint**

The condition of the rule under consideration should evaluate to N/A or error and target should evaluate to true. Formally, \( (\neg rc_i \lor \text{Error}(rc_i)) \land rt_i \)

iii) **Propagation constraint**

If rule under consideration is deny rule, then the mutant is equivalent because the rule level decision in mutant will be deny, and, in original policy, it will be N/A or error. However, for deny-unless-permit RCA, anything other than permit effect results in deny decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both of them will behave the same.

Hence, the rule under consideration should be permit rule such that all the rules with permit effect after the \( i^{th} \) rule should evaluate to N/A or error.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCT when RCA is deny-unless-permit.

\[
(\{PT \lor \text{Error}(PT)\} \land \{\neg rc_i \lor \text{Error}(rc_i)\} \land rt_i \land \{\neg rb_j \lor \text{Error}(rb_i)\} \text{ for any permit rule } r_j = <rb_j, \text{permit}> \text{ for } j \neq i)
\]

c) **first-applicable:**

i) **Reachability constraint**
Policy target should evaluate to \textit{true} or \textit{error} and all the previous rules with permit effect before the current rule under consideration should evaluate to \textit{N/A}.

\textit{ii) Necessity constraint}

If policy target evaluates to \textit{true}, the necessity constraint is that – rule condition evaluates to \textit{N/A} or \textit{error} and rule target evaluates to \textit{true}.

If policy target evaluates to an error, the necessity constraint is that – rule condition evaluates to \textit{N/A} and rule target evaluates to \textit{true}.

\textit{iii) Propagation constraint}

If policy target evaluates to \textit{true} and \(i^{th}\) rule condition evaluates to \textit{false}, then for all other rules which have the same effect as that of \(i^{th}\) rule should evaluate to \textit{N/A} or \textit{error}.

If policy target evaluates to an error and \(i^{th}\) rule condition evaluates to \textit{false}, then for all other rules which have the same effect as that of \(i^{th}\) rule should evaluate to \textit{N/A}.

If policy target evaluates to \textit{true} and \(i^{th}\) rule condition evaluates to an error, then explicit propagation constraint is not required such that reachability and necessity constraint are enough for fault detection.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCT when RCA is \textit{first-applicable}.

\[
(\{PT\} \land \neg rc_i \land rt_i \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \lor Error(rb_j) \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\})
\]

\lor

\[
(Error(PT)) \land \neg rc_i \land rt_i \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\})
\]

\lor
(PT \land Error(rc_i) \land rt_i \land \{\neg rb_j for any rule r_j for j < i\})

3.5 FDC for Rule Condition False (RCF)

Rule condition \textit{false} is a mutation operator which alters the rule condition such that it will always evaluate to \textit{false}. One of the transformation rules to make condition always evaluate to \textit{false} is to introduce new constraint in condition with a random attribute which will always be \textit{false}. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy $P = \langle PT, RCA, RL \rangle$ and $P' = \langle PT, RCA, RL' \rangle$ where $RL = \langle r_1, \ldots, r_i, \ldots, r_n \rangle$, $RL' = \langle r_1, \ldots, r_i', \ldots, r_n \rangle$, $r_i = \langle rt_i, rc_i, re_i \rangle$ and $r_i' = \langle rt_i', rc_i, re_i \rangle$ such that $P'$ is similar to $P$ except the condition $rc_i'$ of $i^{th}$ rule $r_i'$ in $P'$ always evaluates to $false$. Here, $P'$ is called the Rule Condition False (RCF) Mutant of $P$. The fault detection condition for Rule Condition False based on RCA are given below.

a) \textbf{Permit-overrides:}

i) \textit{Reachability constraint}

Policy target should evaluate to \textit{true} or \textit{error} and all the previous rules with permit effect before the current rule under consideration should evaluate to \textit{N/A} or \textit{error} (i.e should not evaluate to \textit{true}).

In formal notation, it will be $\{PT \lor Error(PT)\} \land \{\neg rb_j \lor Error(rb_j) for any permit rule r_j = \langle rb_j, permit \rangle \text{ for } j < i\})

ii) \textit{Necessity constraint}

The necessity constraint is that – rule condition evaluates to \textit{true} or \textit{error} and rule target evaluates to \textit{true}. Formally, $(rc_i \lor Error(rc_i)) \land rt_i$

iii) \textit{Propagation constraint}
If policy target evaluates to true, \(i^{th}\) rule condition evaluates to true and \(i^{th}\) rule effect is deny, then all other rules should evaluate to N/A or error. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is permit rule which evaluates to an error and other is deny rule which evaluates to true or error.

\[
(\{PT\} \land rc_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j)\} \land \forall j \neq i \land \neg(\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, deny>) \land (r_p = <rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d)))
\]

If policy target evaluates to an error, \(i^{th}\) rule condition evaluates to true and \(i^{th}\) rule effect is deny, then all other rules with deny effect should evaluate to N/A and all other rules with permit effect should not evaluate to true. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is permit rule which evaluates to an error and other is deny rule which evaluates to true or error.

\[
(\{Error(PT)\} \land rc_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j)\} \land \forall j \neq i \land \neg(\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = <rb_d, deny>) \land (r_p = <rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d)))
\]

If policy target evaluates to true, \(i^{th}\) rule condition evaluates to true and \(i^{th}\) rule effect is permit, then all other permit rule should evaluate to N/A or error.

\[
(\{PT\} \land rc_i \land r_i = <rb_i, permit> \land \{\neg rb_j \lor Error(rb_j)\} \land \forall j \neq i)
\]

If policy target evaluates to an error, \(i^{th}\) rule condition evaluates to true and \(i^{th}\) rule effect is permit, then all other rules with permit effect should evaluate to N/A if all deny
rules evaluate to N/A. However, if any deny rule evaluates to true or error, then another permit effect can evaluate to N/A or error.

\[
(\{\text{Error}(PT)\} \land rc_i \land r_i = <rb_i, \text{permit}> \land \{-rb_j\ \text{for any rule}\ r_j\ for\ j\ !=\ i\} \lor \{\exists d\ (rb_d \\
\lor \text{Error}(rb_d)) \land (r_d = <rb_d, \text{deny}>)) \land (\neg rb_j \lor \text{Error}(rb_j))\)\ for\ any\ permit\ rule\ r_j = <rb_i, \text{permit}>\ for\ j\ !=\ i\})
\]

If policy target evaluates to true or error, \(i^{th}\) rule condition evaluates to an error and \(i^{th}\) rule effect is deny, then all other rules with deny effect should evaluate to N/A and all other rules with permit effect should evaluate to N/A or error.

\[
(\{\text{PT} \lor \text{Error}(PT)\} \land \text{Error}(rc_i) \land r_i = <rb_i, \text{deny}> \land \{-rb_j\ \lor \text{Error}(rb_j)\)\ for\ any\ permit\ rule\ r_j = <rb_j, \text{permit}>\ for\ j\ !=\ i\} \land \{-rb_i\ \text{for any deny rule}\ r_j = <rb_i, \text{deny}>\ for\ j\ !=\ i\})
\]

If policy target evaluates to true or error, \(i^{th}\) rule condition evaluates to an error and \(i^{th}\) rule effect is permit, then all other rules with permit effect should evaluate to N/A.

\[
(\{\text{PT} \lor \text{Error}(PT)\} \land \text{Error}(rc_i) \land r_i = <rb_i, \text{permit}> \land \{-rb_j\ \lor \text{Error}(rb_j)\)\ for\ any\ permit\ rule\ r_j = <rb_j, \text{permit}>\ for\ j\ !=\ i\})
\]

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCF when RCA is permit-overrides:

\[
(\{\text{PT} \lor rt_i \land rc_i \land r_i = <rb_i, \text{deny}> \land \{-rb_j\ \lor \text{Error}(rb_j)\)\ for\ any\ rule\ r_j\ for\ j\ !=\ i\} \lor \\
\{- (\exists(p,d)\ such\ that\ (i \neq p \neq d)) \land (r_d = <rb_d, \text{deny}>)) \land (r_p = <rb_p, \text{permit}>)) \land \text{Error}(rb_p) \\
\land (\text{Error}(rb_d) \lor rb_d))\})
\]

\[
\lor \\
(\{\text{Error}(PT)\} \land rt_i \land rc_i \land r_i = <rb_i, \text{deny}> \land \{-rb_j\ \lor \text{Error}(rb_j)\)\ for\ any\ permit\ rule\ r_j = <rb_i, \text{permit}>\ for\ j\ !=\ i\} \land \{- (\exists(p,d)\ such\ that\ (i \neq p \neq d)) \land (r_d = <rb_d, \text{deny}>)) \land (r_p = <rb_p, \text{permit}>)) \land \text{Error}(rb_p) \\
\land (\text{Error}(rb_d) \lor rb_d))\})
\]
such that \((i \neq p \neq d) \land (r_d = <rb_d, deny>) \land (r_p = <rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d))\)
\[
\lor
\]
\((\{PT\} \land rt_i \land rc_i \land r_i = <rb_i, permit> \land \{\neg rb_j \lor Error(rb_j)\} \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \neq i))\)
\[
\lor
\]
\((\{Error(PT)\} \land rt_i \land rc_i \land r_i = <rb_i, permit> \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j \neq i\} \lor \{\exists d (rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \neq i \}))\)
\[
\lor
\]
\((\{PT \lor Error(PT)\} \land Error(rc_i) \land rt_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j)\} \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \neq i\} \land \{\neg rb_j \text{ for any deny rule } r_j = <rb_j, deny> \text{ for } j \neq i\})\)
\[
\lor
\]
\((\{PT \lor Error(PT)\} \land Error(rc_i) \land rt_i \land r_i = <rb_i, permit> \land \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \neq i\})\)

**b) deny-unless-permit:**

1) **Reachability constraint**

The policy target should be *true or should evaluate to an error*. For any *permit rule* \(r_j = <rb_j, permit> (j<i)\) before rule \(r_i\), \(rb_j\) should not evaluate to *true* (i.e. it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with *permit* effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e. should not evaluate to *true*).
In formal notation, it will be \( \{PT \lor Error(PT)\} \land \{\neg rb_j \lor Error(rb_j)\} \) for any permit rule \( r_j = <rb_j, \text{permit}> \) for \( j < i \)

ii) **Necessity constraint**

The target of the rule under consideration should evaluate to true and condition should evaluate to true. Formally, \( rt_i \land rc_i \)

iii) **Propagation constraint**

If rule under consideration is deny rule, then the mutant is equivalent because the rule level decision in mutant will be N/A and, in original policy, it will be true or error. However, for deny-unless-permit RCA, anything other than permit effect results in deny decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both will behave the same.

Hence, the rule under consideration should be permit rule such that all the rules with permit effect evaluate to N/A or error.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCF when RCA is deny-unless-permit.

\[
(\{PT \lor Error(PT)\} \land \{rt_i \land rc_i\} \land \{\neg rb_j \lor Error(rb_j)\} \text{ for any permit rule } r_j = <rb_j, \text{permit} > \text{ for } j \neq i)
\]

c) **first-applicable:**

i) **Reachability constraint**

Policy target should evaluate to true or error and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A.

ii) **Necessity constraint**
If policy target evaluates to \textit{true}, the necessity constraint is that – rule condition evaluates to \textit{N/A} or \textit{error} and rule target evaluates to \textit{true}.

If policy target evaluates to an error, the necessity constraint is that – rule condition evaluates to \textit{N/A} and rule target evaluates to \textit{true}.

\textit{iii) Propagation constraint}

If policy target evaluates to \textit{true} and \(i\)\textsuperscript{th} rule condition evaluates to \textit{true}, then for all other rules which have the same effect as that of \(i\)\textsuperscript{th} rule should evaluate to \textit{N/A} or \textit{error}.

If policy target evaluates to \textit{true} and \(i\)\textsuperscript{th} rule condition evaluates to an error, then for all other rules after the \(i\)\textsuperscript{th} rule which has the same effect as that of \(i\)\textsuperscript{th} rule should evaluate to \textit{N/A} or \textit{true}.

If policy target evaluates to an error and \(i\)\textsuperscript{th} rule condition evaluates to an error or \textit{true}, then for all other rules after the \(i\)\textsuperscript{th} rule which has the same effect as that of \(i\)\textsuperscript{th} rule should evaluate to \textit{N/A}.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCF when RCA is \textit{first-applicable}.

\[
\left(\{PT\} \land rt_i \land rc_i \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \lor Error(rb_j) \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\}\right)
\lor
\[
\left(\{PT\} \land Error(rc_i) \land rt_i \land \{\neg rb_j \lor rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\}\right)
\lor
\[
\left(\{Error(PT)\} \land (rc_i \lor Error(rc_i)) \land rt_i \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\}\right)
\]
3.6 FDC for Add Not Function (ANF)

Add not function is a mutation operator which adds a not function to the rule condition such that rule condition will always evaluate to false on faulty policy if it evaluates to true in original policy, and vice versa. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy $P = \langle PT, RCA, RL \rangle$ and $P' = \langle PT, RCA, RL' \rangle$ where $RL = \langle r_1, \ldots, r_i, \ldots, r_n \rangle$, $RL' = \langle r_1, \ldots, r_i', \ldots, r_n \rangle$, $r_i = \langle rt_i, rc_i, re_i \rangle$ and $r_i' = \langle rt_i', rc_i, re_i \rangle$ such that $P'$ is similar to $P$ except the condition $rc_i'$ of $i^{th}$ rule $r_i'$ in $P'$ is $\neg rc_i$. Here, $P'$ is called the Add Not Function (ANF) Mutant of $P$. The fault detection condition for ANF based on RCA are given below.

a) Permit-overrides:
   i) Reachability constraint

   Policy target should evaluate to true or error and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A or error (i.e. should not evaluate to true).

   In formal notation, it will be $\{PT \lor Error(PT)\} \land \{\neg rb_j \lor Error(rb_j)\}$ for any permit rule $r_j = \langle rb_j, permit \rangle$ for $j < i$.

   ii) Necessity constraint

   The necessity constraint is that – rule condition evaluates to true and rule target evaluates to true. Formally, $rc_i \land rt_i$

   Note: we do not consider the error case on condition because error case will produce no different intermediate results between faulty and original policy.

   iii) Propagation constraint
If policy target evaluates to true, \(i^{th}\) rule condition evaluates to true and \(i^{th}\) rule effect is deny, then all other rules should evaluate to N/A or error. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is permit rule which evaluates to an error and other is deny rule which evaluates to true or error.

\[
(\{PT\} \land rc_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) \text{ for any rule } r_j \neq i\} \land \neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = < rb_d, deny>) \land (r_p = < rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d)))
\]

If policy target evaluates to an error, \(i^{th}\) rule condition evaluates to true and \(i^{th}\) rule effect is deny, then all other rules with deny effect should evaluate to N/A and all other rules with permit effect should not evaluate to true. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is permit rule which evaluates to an error and other is deny rule which evaluates to true or error.

\[
(\{Error(PT)\} \land rc_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \neq i\} \land \neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = < rb_d, deny>) \land (r_p = < rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d)))
\]

If policy target evaluates to true, \(i^{th}\) rule condition evaluates to true and \(i^{th}\) rule effect is permit, then all other permit rule should evaluate to N/A or error.

\[
(\{PT\} \land rc_i \land r_i = <rb_i, permit> \land \{\neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \neq i\})
\]

If policy target evaluates to an error, \(i^{th}\) rule condition evaluates to true and \(i^{th}\) rule effect is permit, then all other rules with permit effect should evaluate to N/A if all deny
rules evaluate to N/A. However, if any deny rule evaluates to true or error, then another permit effect can evaluate to N/A or error.

\[
\{\text{Error}(PT)\} \land r_{ci} \land r_i = \langle rb_i, \text{permit}\rangle \land \{\neg r_j \text{ for any rule } r_j \text{ for } j \neq i\} \lor \{\exists d \ (rb_d \\
\lor \text{Error}(rb_d)) \land (r_d = \langle rb_d, \text{deny}\rangle) \land (\neg r_j \lor \text{Error}(rb_j)) \text{ for any permit rule } r_j = \langle rb_i, \text{permit}\rangle \text{ for } j \neq i}\}
\]

Combining all constraints as a single constraint, we get the following constraint for fault detection of ANF when RCA is permit-overrides:

\[
\{\text{PT}\} \land rt_i \land rc_i \land r_i = \langle rb_i, \text{deny}\rangle \land \{\neg r_j \lor \text{Error}(rb_j) \text{ for any permit rule } r_j = \langle rb_i, \text{permit}\rangle \text{ for } j \neq i\} \land \{\neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = \langle rb_d, \text{deny}\rangle) \land (r_p = \langle rb_p, \text{permit}\rangle) \land \text{Error}(rb_p) \land (\text{Error}(rb_d) \lor rb_d)))\}
\]

\lor

\[
\{\text{Error}(PT)\} \land rt_i \land rc_i \land r_i = \langle rb_i, \text{deny}\rangle \land \{\neg r_j \lor \text{Error}(rb_j) \text{ for any deny rule } r_j = \langle rb_i, \text{deny}\rangle \text{ for } j \neq i\} \land \{\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = \langle rb_d, \text{deny}\rangle) \land (r_p = \langle rb_p, \text{permit}\rangle) \land \text{Error}(rb_p) \land (\text{Error}(rb_d) \lor rb_d)))\}
\]

\lor

\[
\{\text{PT}\} \land rt_i \land rc_i \land r_i = \langle rb_i, \text{permit}\rangle \land \{\neg r_j \lor \text{Error}(rb_j) \text{ for any permit rule } r_j = \langle rb_i, \text{permit}\rangle \text{ for } j \neq i\}
\]

\lor

\[
\{\text{Error}(PT)\} \land rt_i \land rc_i \land r_i = \langle rb_i, \text{permit}\rangle \land \{\neg r_j \text{ for any rule } r_j \text{ for } j \neq i\} \lor \{\exists d (rb_d \lor \text{Error}(rb_d)) \land (r_d = \langle rb_d, \text{deny}\rangle) \land (\neg r_j \lor \text{Error}(rb_j)) \text{ for any permit rule } r_j = \langle rb_i, \text{permit}\rangle \text{ for } j \neq i\}
\]

b) **deny-unless-permit:**

i) **Reachability constraint**

The policy target should be *true* or should evaluate to an error. For any *permit* rule $r_j$ $= <rb_j, permit> \ (j<i)$ before rule $r_i$, $rb_j$ should not evaluate to *true* (i.e it should be N/A or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with *permit* effect before the current rule under consideration should evaluate to N/A or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \lor Error(PT)\} \land \{\neg rb_j \lor Error(rb_j)\}$ for any *permit* rule $r_j = <rb_j, permit> \ for \ j < i$

ii) **Necessity constraint**

The target of the rule under consideration should evaluate to *true* and condition should evaluate to *true*. Formally, $rt_i \land rc_i$

iii) **Propagation constraint**

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be N/A, and, in original policy, it will be *true* or *error*. 

However, for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect evaluate to N/A or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of ANF when RCA is *deny-unless-permit*. 
((PT ∨ Error(PT)) ∧ {rtᵢ ∧ rcᵢ} ∧ {¬rbⱼ ∨ Error(rbⱼ) for any rule rⱼ = < rbⱼ, permit for j ≠ i})

c) first-applicable:

i) Reachability constraint

Policy target should evaluate to true or error and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A.

ii) Necessity constraint

The necessity constraint is that – rule condition evaluates to N/A and rule target evaluates to true.

iii) Propagation constraint

If policy target evaluates to true and iᵗʰ rule condition evaluates to true, then for all other rules which have the same effect as that of iᵗʰ rule should evaluate to N/A or error.

If policy target evaluates to an error and iᵗʰ rule condition evaluates true, then for all other rules after the iᵗʰ rule which has the same effect as that of iᵗʰ rule should evaluate to N/A.

Combining all constraints as a single constraint, we get the following constraint for fault detection of ANF when RCA is first-applicable.

\((\{PT \lor Error(PT)\} \land \{rtᵢ \land rcᵢ\} \land \{¬rbⱼ \lor Error(rbⱼ)\}\ for any rule rⱼ = < rbⱼ, permit for j ≠ i}\))

\lor

\((\{Error(PT)\} \land rcᵢ \land rtᵢ \land \{¬rbⱼ \lor Error(rbⱼ)\}\ for any rule rⱼ such that reᵢ = reⱼ for j > i})

3.7 FDC for Remove Not Function (RNF)
Remove not function is a mutation operator which removes a not function from the rule condition such that rule condition will always evaluate to false on faulty policy if it evaluates to true in original policy, and vice versa. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy \( P = \langle PT, RCA, RL \rangle \) and \( P' = \langle PT, RCA, RL' \rangle \) where \( RL = \langle r_1, ..., r_i, ..., r_n \rangle \), \( RL' = \langle r_1, ..., r'_i, ..., r_n \rangle \), \( r_i = \langle rt_i, rc_i, re_i \rangle \) and \( r'_i = \langle rt'_i, rc_i, re_i \rangle \) such that \( P' \) is similar to \( P \) except the condition \( rc_i' \) of \( i^{th} \) rule \( r_i \) in \( P' \) is \( \neg rc_i \). Here, \( P' \) is called the Remove Not Function (RNF) Mutant of \( P \). The fault detection condition for RNF is same as that for ANF because the effect of mutation operator on both cases is \( rc_i' = \neg rc_i \).

3.8 FDC for Remove a Rule (RER)

Remove a rule is a mutation operator which removes a rule from the policy such that rule condition will always evaluate to false on faulty policy if it evaluates to true in original policy, and vice versa. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy \( P = \langle PT, RCA, RL \rangle \) and \( P' = \langle PT, RCA, RL' \rangle \) where \( RL = \langle r_1, ..., r_i, ..., r_n \rangle \), \( RL' = \langle r_1, ..., r_{i-1}, r_{i+1}, ..., r_n \rangle \), \( r_i = \langle rt_i, rc_i, re_i \rangle \) and \( r'_i = \langle rt'_i, rc_i, re_i \rangle \) such that \( P' \) is similar to \( P \) except for that \( P' \) does not contain \( r_i \) in in rule list. Here, \( P' \) is called the Remove a Rule (RER) Mutant of \( P \). The fault detection condition for RER based on RCA are given below.

\[ \text{a) Permit-overrides:} \]
\[ \text{i) Reachability constraint} \]
Policy target should evaluate to true or error and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A or error (i.e. should not evaluate to true).

In formal notation, it will be \{PT ˅ Error(PT)\} ˄ \{-rbj ˅ Error(rbj)\} for any permit rule \(r_j = <rb_j, permit>\) for \(j < i\)

ii) Necessity constraint

The necessity constraint is that – rule body evaluates to true or error. Formally, \(rb_i ˅ Error(rb_i)\)

iii) Propagation constraint

If policy target evaluates to true, \(i^{th}\) rule body evaluates to true and \(i^{th}\) rule effect is deny, then all other rules should evaluate to N/A or error. Further, there should not exist a pair of rules (excluding the current rule under consideration) such that one of them is permit rule which evaluates to an error and other is deny rule which evaluates to true or error.

\((\{PT\} ˄ rb_i ˄ ri = <rb_i, deny> ˄ \{-rb_j ˅ Error(rb_j)\} for any rule \(r_j\) for \(j \neq i\) ˄ \{-∃(p,d) such that \((i \neq p \neq d) ˄ (rd = <rb_d, deny>) ˄ (rp = <rb_p, permit>) ˄ Error(rb_p) ˄ (Error(rb_d) ˅ rb_d))\})\)

If policy target evaluates to an error, \(i^{th}\) rule body evaluates to true and \(i^{th}\) rule effect is deny, then all other rules with deny effect should evaluate to N/A and all other rules with permit effect should not evaluate to true. Further, there should not exist a pair of rules (excluding the current rule under consideration) such that one of them is permit rule which evaluates to an error and other is deny rule which evaluates to true or error.

\((\{Error(PT)\} ˄ rb_i ˄ ri = <rb_i, deny> ˄ \{-rb_j ˅ Error(rb_j)\} for any permit rule \(r_j = <rb_j, permit>\) for \(j \neq i\) ˄ \{-∃(p,d) such
that \((i \neq p \neq d) \land (r_d = <rb_d, deny>) \land (r_p = <rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d))\)

If policy target evaluates to \textit{true}, \(i^{th}\) rule body evaluates to \textit{true} and \(i^{th}\) rule effect is \textit{permit}, then all other permit rule should evaluate to \textit{N/A} or \textit{error}.

\[
\{PT \} \land rb_i \land r_i = <rb_i, permit> \land \{\neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \neq i\}
\]

If policy target evaluates to an error, \(i^{th}\) rule body evaluates to \textit{true} and \(i^{th}\) rule effect is \textit{permit}, then all other rules with \textit{permit} effect should evaluate to \textit{N/A} if all \textit{deny} rules evaluate to \textit{N/A}. However, if any \textit{deny} rule evaluates to \textit{true} or \textit{error}, then another permit effect can evaluate to \textit{N/A} or \textit{error}.

\[
\{Error(PT)\} \land rb_i \land r_i = <rb_i, permit> \land \{\neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \neq i\} \lor \{\exists d ( rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \neq i\}
\]

If policy target evaluates to \textit{true or error}, \(i^{th}\) rule body evaluates to an error and \(i^{th}\) rule effect is \textit{deny}, then all other rules with \textit{deny} effect should evaluate to \textit{N/A} and all other rules with \textit{permit} effect should evaluate to \textit{N/A} or \textit{error}.

\[
\{PT \lor Error(PT)\} \land Error(rb_i) \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \neq i\} \land \{\neg rb_j \text{ for any deny rule } r_j = <rb_j, deny> \text{ for } j \neq i\}
\]

If policy target evaluates to \textit{true or error}, \(i^{th}\) rule body evaluates to an error and \(i^{th}\) rule effect is \textit{permit}, then all other rules with \textit{permit} effect should evaluate to \textit{N/A}.

\[
\{PT \lor Error(PT)\} \land Error(rb_i) \land r_i = <rb_i, permit> \land \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \neq i\}
\]
Combining all constraints as a single constraint, we get the following constraint for fault detection of RER when RCA is *permit overrides*:

\[
\begin{align*}
&\{(PT) \land rb_1 \land r_i = <rb_i, deny> \land \neg rb_j \lor Error(rb_j) \text{ for any rule } r_j \neq i \} \land \neg( \exists(p,d) \\
&\text{such that } (i \neq p \neq d) \land (r_d = <rb_d, deny>) \land (r_p = <rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d))
\end{align*}
\]

\[
\lor
\]

\[
\begin{align*}
&\{(Error(PT)) \land rb_i \land r_i = <rb_i, deny> \land \neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, \\
&\text{permit for } j \neq i \} \land \neg rb_j \lor Error(rb_j) \text{ for } j \neq i \} \land \neg( \exists(p,d) \\
&\text{such that } (i \neq p \neq d) \land (r_d = <rb_d, deny>) \land (r_p = <rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d))
\end{align*}
\]

\[
\lor
\]

\[
\begin{align*}
&\{(PT) \land rb_i \land r_i = <rb_i, permit> \land \neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, \\
&\text{permit for } j \neq i \} \land \neg rb_j \lor Error(rb_j) \text{ for } j \neq i \} \land \exists d (rb_d \lor \\
&Error(rb_d)) \land (r_d = <rb_d, deny>) \land \neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, \\
&\text{permit for } j \neq i )
\end{align*}
\]

\[
\lor
\]

\[
\begin{align*}
&\{(PT \lor Error(PT)) \land Error(rb) \land r_i = <rb_i, deny> \land \neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, \\
&\text{permit for } j \neq i \} \land \neg rb_j \lor Error(rb_j) \text{ for } j \neq i \} \land \neg rb_j \lor Error(rb_j) \text{ for } j \neq i \} \land \exists d (rb_d \lor \\
&Error(rb_d)) \land (r_d = <rb_d, deny>) \land \neg rb_j \lor Error(rb_j) \text{ for any perm} \\
&\text{it rule } r_j = <rb_j, permit> \text{ for } j \neq i \}
\end{align*}
\]

\[
\lor
\]

\[
\begin{align*}
&\{(PT \lor Error(PT)) \land Error(rb) \land r_i = <rb_i, permit> \land \neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_j, \\
&\text{permit for } j \neq i \}
\end{align*}
\]
b) *deny-unless-permit:*

i) *Reachability constraint*

The policy target should be *true* or should evaluate to an error. For any *permit* rule \( r_j \) = \( \langle rb_j, \text{permit} \rangle \) (\( j < i \)) before rule \( r_i \), \( rb_j \) should not evaluate to *true* (i.e it should be N/A or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rule with *permit* effect before the current rule under consideration should evaluate to N/A or *error* (i.e should not evaluate to *true*).

In formal notation, it will be \( \{PT \lor \text{Error}(PT)\} \land \{ \neg rb_j \lor \text{Error}(rb_j) \} \) for any *permit* rule \( r_j = \langle rb_j, \text{permit} \rangle \) for \( j < i \)

ii) *Necessity constraint*

The rule should evaluate to *true*. Formally, \( rb_i \)

iii) *Propagation constraint*

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be N/A and, in original policy, it will be *true* or *error*. However, for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect evaluate to N/A or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RER when RCA is *deny-unless-permit*. 
\[(PT \lor Error(PT)) \land \{rb_i\} \land \{\neg rb_j \lor Error(rb_i)\} \text{ for any permit rule } r_j = < rb_j, \text{ permit } > \text{ for } j \neq i)\]

c) \textit{first-applicable:}

i) \textit{Reachability constraint}

Policy target should evaluate to \textit{true} or \textit{error} and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A.

ii) \textit{Necessity constraint}

If policy target evaluates to \textit{true}, the necessity constraint is that \textit{rule body} evaluates to N/A or \textit{error}.

If policy target evaluates to an error, the necessity constraint is that \textit{rule body} evaluates to N/A.

iii) \textit{Propagation constraint}

If policy target evaluates to \textit{true} and \textit{i}th rule body evaluates to \textit{true}, then for all other rules which have the same effect as that of \textit{i}th rule should evaluate to N/A or \textit{error}.

If policy target evaluates to \textit{true} and \textit{i}th rule body evaluates to an error, then for all other rules after the \textit{i}th rule which has the same effect as that of \textit{i}th rule should evaluate to N/A or \textit{true}.

If policy target evaluates to an error and \textit{i}th rule body evaluates to an error or \textit{true}, then for all other rules after the \textit{i}th rule which has the same effect as that of \textit{i}th rule should evaluate to N/A.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RER when RCA is \textit{first-applicable}.
\[({PT} \land rb_i \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \lor \text{Error}(rb_j) \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\})
\]
\[\lor\]
\[({PT} \land \text{Error}(rb_i) \land \{\neg rb_j \lor rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\})
\]
\[\lor\]
\[({\text{Error}(PT) \land (rb_i \lor \text{Error}(rb_i)) \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \land \{\neg rb_j \text{ for any rule } r_j \text{ such that } re_i = re_j \text{ for } j > i\})
\]

3.9 FDC for Policy Target True (PTT)

Policy target true is a mutation operator which alters the policy target such that it will always evaluate to true. One of the transformation rules to make target always evaluate to true is to make it empty so that it will always evaluate to true. Since it has a fault in the target of a policy, it is under the category incorrect policy target.

Consider an XACML policy \(P = <PT, RCA, RL>\) and \(P' = <PT', RCA, RL'>\) where \(PT\) and \(PT'\) are such that \(P'\) is similar to \(P\) except the target \(PT'\) of \(P'\) always evaluates to true. Here, \(P'\) is called the Policy Target True (PTT) Mutant of \(P\). The fault detection condition for Policy Target True based on RCA are given below.

a) permit-overrides:

i) Reachability constraint

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) Necessity constraint

The policy target evaluates to N/A or error.
iii) **Propagation constraint**

If policy target evaluates to N/A, at least one rule should evaluate to *true* or *error*.

If policy target evaluates to an error, there should be one permit rule that evaluates to *true* or all permit rules evaluate to N/A and at least one deny rule evaluates to *true*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is *permit-overrides*:

\[
(\neg PT \land \{rb_j \lor Error(rb_j) \forall any rule r_j\} \lor (Error(PT) \land \{rb\forall any rule r_j such that r_j

\]

\[= rb_j, permit\}) \lor \{\neg rb_j for all rule r_j such that r_j = rb_j, permit}\land rb\forall any rule r_j

\]

\[such that r_j = rb_j, deny\} ))

b) **deny-unless-permit:**

i) **Reachability constraint**

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) **Necessity constraint**

The policy target evaluates to N/A or *error*.

iii) **Propagation constraint**

Necessity constraint is sufficient for fault detection. Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is *deny-unless-permit*:  

\[
\neg PT \lor Error(PT)
\]

c) **first-applicable:**

i) **Reachability constraint**

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.
ii) **Necessity constraint**

The policy target evaluates to N/A or error.

iii) **Propagation constraint**

If policy target evaluates to true, at least one rule should evaluate to true or error.

If policy target evaluates to an error, there should be one rule that evaluates to true and all other rules before that rule should evaluate to N/A.

Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is first-applicable:

\[
(\neg PT \land \{rb_j \lor \text{Error}(rb_j) \text{ for any rule } r_j\}) \lor (\text{Error}(PT) \land (\{rb_j \text{ for any rule } r_j \land \neg rb_i \text{ for any rule } r_i \text{ such that } i < j\})
\]

3.10 FDC for Policy Target False (PTF)

Policy target false is a mutation operator which alters the policy target such that it will always evaluate to false. Since it has a fault in the target of a policy, it is under the category incorrect policy target.

Consider an XACML policy \( P = <PT, RCA, RL> \) and \( P' = <PT', RCA, RL'> \) where \( PT \) and \( PT' \) are such that \( P' \) is similar to \( P \) except the target \( PT' \) of \( P' \) always evaluates to false. Here, \( P' \) is called the Policy Target False (PTF) Mutant of \( P \). The fault detection condition for Policy Target False based on RCA are given below.

a) **permit-overrides:**

i) **Reachability constraint**

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) **Necessity constraint**
The policy target evaluates to *true* or *error*.

### iii) Propagation constraint

At least one rule should evaluate to *true* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of PTF when RCA is *permit-overrides*:

\[ \neg PT \lor \text{Error}(PT) \land \{ rb_j \lor \text{Error}(rb_j) \text{ for any rule } r_j \} \]

b) *deny-unless-permit:*

i) **Reachability constraint**

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) **Necessity constraint**

The policy target evaluates to *true* or *error*.

iii) **Propagation constraint**

Necessity constraint is sufficient for fault detection. Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is *permit-overrides*:

\[ PT \lor \text{Error}(PT) \]

c) **first-applicable:**

i) **Reachability constraint**

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) **Necessity constraint**

The policy target evaluates to *true* or *error*.

iii) **Propagation constraint**
At least one rule should evaluate to \textit{true} or \textit{error}.

Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is \textit{permit-override}:
\[
((\neg \text{PT} \lor \text{Error(PT)}) \land \{\text{rb}_j \lor \text{Error(rb}_j)\text{ for any rule } r_j\})
\]

3.11 FDC for First Permit Rule (FPR)

First Permit Rule is a mutation operator which changes the position of the first Deny rule with the first Permit rule that follows the first Deny rule. Since it has a fault with the ordering of the rules, it is under the category incorrect rule ordering.

Consider an XACML policy \(P = \langle \text{PT}, \text{RCA}, \text{RL} \rangle\) where \(\text{RL} = \langle r_1, \ldots, r_d \ldots r_p \ldots r_n \rangle\), \(\text{RL'} = \langle r_1, \ldots, r_p, \ldots r_d, \ldots, r_n \rangle\) such that \(P'\) is similar to \(P\) except the order of first permit rule following the first deny rule in \(P\) is swapped in \(P'\).

\textbf{Note: The order of permit rules and deny rules does not alter the decision of the policy element if the RCA are Permit-Override, Deny Override, Permit-unless-Deny or Deny-unless-Permit}

The order of permit rules and deny rules matters only when both or at least one pair of permit rule and deny rule can be applied (true or error) at the same time. If they cannot be applied at the same time, then such a mutant will be equivalent to the original policy.

The fault detection condition of FPR when RCA is \textit{first-applicable} is as follows:

\begin{enumerate}

\item \textit{Reachability constraint}

The policy target evaluates to \textit{true} or \textit{error}.

\item \textit{Necessity constraint}

The rule with first \textit{permit} effect should evaluate to \textit{true} or \textit{error} and the rule with first \textit{deny} effect following the first \textit{permit} rule should evaluate to \textit{true} or \textit{error}.

\item \textit{Propagation constraint}

\end{enumerate}
Necessity constraint is sufficient for fault detection.

\[(\text{rb}_p \vee \text{Error(rb}_p)) \land (\text{rb}_d \vee \text{Error(rb}_d))\] for the first permit rule \(r_p = <\text{rb}_p, \text{permit}>\) and for the first deny rule \(r_d = <\text{rb}_d, \text{deny}>\) after \(r_p\).

### 3.12 FDC for First Deny Rule (FDR)

First Deny Rule is a mutation operator which changes the position of the first Permit rule with the first Deny rule that follows the first Permit rule. Since it has a fault with the ordering of the rules, it is under the category incorrect rule ordering.

Consider an XACML policy \(P = \langle\text{PT}, \text{RCA}, \text{RL}\rangle\) where \(\text{RL} = <r_1, ..., r_d, ..., r_p, ..., r_n>\), \(\text{RL'} = <r_1, ..., r_p, ..., r_d, ..., r_n>\) such that \(P'\) is similar to \(P\) except the order of first deny rule following the first permit rule in \(P\) is swapped in \(P'\).

The fault detection condition of FDR when RCA is *first-applicable* is as follows:

**i)** *Reachability constraint*

The policy target evaluates to *true* or *error*.

**ii)** *Necessity constraint*

The rule with first *deny* effect should evaluate to *true* or *error* and the rule with first *permit* effect following the first *deny* rule should evaluate to *true* or *error*.

**iii)** *Propagation constraint*

Necessity constraint is sufficient for fault detection.

\[(\text{rb}_d \vee \text{Error(rb}_d)) \land (\text{rb}_p \vee \text{Error(rb}_p))\] for the first deny rule \(r_d = <\text{rb}_d, \text{deny}>\) and for the first permit rule \(r_p = <\text{rb}_p, \text{permit}>\) after \(r_d\).

### 3.13 FDC for Remove Parallel Target Element (RPTE)
Remove Parallel Target Element is a mutation operator in which an \textit{AnyOf} element of the target or \textit{AllOf} element of an \textit{AnyOf} element is removed from the target. Since it has a fault of missing a target element, it is under the category missing target element.

Let PT be the target element of Policy P of the rule \( r_i \). The target element is composed of the conjunction of \textit{AnyOf} element and each \textit{AnyOf} element is composed of the disjunction of \textit{AllOf} elements. Hence \( PT = \text{AnyOf}_1 \land \text{AnyOf}_2 \land \ldots \text{AnyOf}_i \land \ldots \text{AnyOf}_n \) and arbitrary \( i^{th} \text{ AnyOf}_i = \text{AllOf}_{i1} \lor \text{AllOf}_{i2} \lor \ldots \text{AllOf}_{im} \). If either \( i^{th} \text{ AnyOf} \) is removed or \( j^{th} \text{ AllOf} \) from \( i^{th} \text{ AnyOf} \) is removed, the resulted faulty policy is called Remove parallel target element mutant. The fault detection condition for RPTE is presented as follows:

\[
\text{a) Permit-overrides:}
\]

\[i) \text{ Reachability constraint} \]

Policy target should evaluate to \textit{true} or \textit{error} and all the previous rules with permit effect before the current rule under consideration should evaluate to \textit{N/A} or \textit{error} (i.e should not evaluate to \textit{true}).

In formal notation, it will be \( \{PT \lor Error(PT)\} \land \{\neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j < i\} \)

\[ii) \text{ Necessity constraint} \]

The necessity constraint for RPTE with \( i^{th} \text{ AnyOf} \) element missing is that – rule target expression should have negation term for \( i^{th} \text{ AnyOf} \) element and rule condition evaluates to \textit{true}.

Formally, \( (\text{AnyOf}_1 \land \neg \text{AnyOf}_i \land \ldots \text{AnyOf}_n) \land rc_i \)

The necessity constraint for RPTE with \( j^{th} \text{ AllOf} \) element of \( i^{th} \text{ AnyOf} \) element missing is that – rule target expression should have a positive term for \( j^{th} \text{ AllOf} \) element of \( i^{th} \text{ AnyOf} \).
element and all other \textit{AllOf} element should have a negative term and rule condition evaluates to \textit{true}.

Formally, $(\text{AnyOf}_i \land \text{AnyOf}_i = (\neg \text{AllOf}_i \lor \text{AllOf}_j \lor \neg \text{AllOf}_m) \land .. \text{AnyOf}_n) \land rc_i$

\textbf{iii) Propagation constraint}

If policy target evaluates to \textit{true} and \textit{i}th rule effect is \textit{deny}, then all other rules should evaluate to \textit{N/A} or \textit{error}. Further, there should not exist a pair of rules (excluding the current rule under consideration) such that one of them is \textit{permit} rule which evaluates to an error and other is \textit{deny} rule which evaluates to \textit{true} or \textit{error}.

$(\{\text{PT}\} \land r_i = <\text{rb}_i, \text{deny}> \land \{\neg \text{rb}_j \lor \text{Error(\text{rb}_j)} \text{for any rule } r_j \text{ for } j \neq i\} \land \{\neg (\exists (p, d) \text{ such that } (i \neq p \neq d) \land (r_d = <\text{rb}_d, \text{deny}>)) \land (r_p = <\text{rb}_p, \text{permit}>)) \land \text{Error(\text{rb}_p)} \land (\text{Error(\text{rb}_d)} \lor \text{rb}_d)\}))$

If policy target evaluates to an error, and \textit{i}th rule effect is \textit{deny}, then all other rules with \textit{deny} effect should evaluate to \textit{N/A} and all other rules with \textit{permit} effect should not evaluate to \textit{true}. Further, there should not exist a pair of rules (excluding the current rule under consideration) such that one of them is \textit{permit} rule which evaluates to an error and other is \textit{deny} rule which evaluates to \textit{true} or \textit{error}.

$(\{\text{Error(PT)}\} \land r_i = <\text{rb}_i, \text{deny}> \land \{\neg \text{rb}_j \lor \text{Error(\text{rb}_j)} \text{for any permit rule } r_j = <\text{rb}_i, \text{permit}> \text{ for } j \neq i\} \land \{\neg (\exists (p, d) \text{ such that } (i \neq p \neq d) \land (r_d = <\text{rb}_d, \text{deny}>)) \land (r_p = <\text{rb}_p, \text{permit}>)) \land \text{Error(\text{rb}_p)} \land (\text{Error(\text{rb}_d)} \lor \text{rb}_d)\}))$

If policy target evaluates to \textit{true}, and \textit{i}th rule effect is \textit{permit}, then all other permit rules should evaluate to \textit{N/A} or \textit{error}.

$(\{\text{PT}\} \land r_i = <\text{rb}_i, \text{permit}> \land \{\neg \text{rb}_j \lor \text{Error(\text{rb}_j)} \text{for any permit rule } r_j = <\text{rb}_i, \text{permit}> \text{ for } j \neq i\})$
If policy target evaluates to an error, and $i^{th}$ rule effect is permit, then all other rules with permit effect should evaluate to N/A if all deny rules evaluate to N/A. However, if any deny rule evaluates to true or error, then another permit effect can evaluate to N/A or error.

\[
(\text{Error}(PT)} \land r_i = <\text{rb}_b, \text{permit}> \land \{\neg \text{rb}_j \text{ for any rule } r_j \neq i\} \lor \{\exists d \ (\text{rb}_d \lor \text{Error(\text{rb}_d)}) \land (r_d = <\text{rb}_d, \text{deny}> \land \neg \text{rb}_j \lor \text{Error(\text{rb}_j)})) \text{ for any permit rule } r_j = <\text{rb}_i, \\
\text{permit}> \text{ for } j \neq i\})
\]

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is permit-overrides:

\[
(\text{PT } \land \{\text{AnyOf}_1 \land \neg \text{AnyOf}_i \land .. \text{AnyOf}_n\} \lor (\text{AnyOf}_1 \land \text{AnyOf}_i = (\neg \text{AllOf}_1 \lor \text{AllOf}_j \lor \neg \text{AllOf}_m) \land .. \text{AnyOf}_n)) \land r_c \land r_i = <\text{rb}_i, \text{deny}> \land \{\neg \text{rb}_j \lor \text{Error(\text{rb}_j)} \text{ for any rule } r_j \neq i\} \land \{\neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = <\text{rb}_d, \text{deny}>)) \land (r_p = <\text{rb}_p, \text{permit}>) \land \\
\text{Error(\text{rb}_p) \lor (\text{Error(\text{rb}_d) \lor \text{rb}_d}))}
\]

\[
\lor
\]

\[
(\text{Error(PT)} \land \{\text{AnyOf}_1 \land \neg \text{AnyOf}_i \land .. \text{AnyOf}_n\} \lor (\text{AnyOf}_1 \land \text{AnyOf}_i = (\neg \text{AllOf}_1 \lor \text{AllOf}_j \lor \neg \text{AllOf}_m) \land .. \text{AnyOf}_n)) \land r_c \land r_i = <\text{rb}_i, \text{deny}> \land \{\neg \text{rb}_j \lor \text{Error(\text{rb}_j)} \text{ for any permit rule } r_j = <\text{rb}_i, \text{permit}> \text{ for } j \neq i\} \land \{\neg r_j \text{ for any deny rule } r_j = <\text{rb}_i, \text{deny}> \text{ for } j \neq i\} \land \{\neg (\exists (p,d) \text{ such that } (i \neq p \neq d) \land (r_d = <\text{rb}_d, \text{deny}>) \land (r_p = <\text{rb}_p, \text{permit}>) \land \\
\text{Error(\text{rb}_p) \lor (\text{Error(\text{rb}_d) \lor \text{rb}_d}))}
\]

\[
\lor
\]

\[
(\text{PT } \land \{\text{AnyOf}_1 \land \neg \text{AnyOf}_i \land .. \text{AnyOf}_n\} \lor (\text{AnyOf}_1 \land \text{AnyOf}_i = (\neg \text{AllOf}_1 \lor \text{AllOf}_j \lor \neg \text{AllOf}_m) \land .. \text{AnyOf}_n)) \land r_c \land r_i = <\text{rb}_i, \text{permit}> \land \{\neg \text{rb}_j \lor \text{Error(\text{rb}_j)} \text{ for any permit rule } r_j = <\text{rb}_i, \text{permit}> \text{ for } j \neq i\})
\]

\[
\lor
\]
b) **deny-unless-permit:**

i) **Reachability constraint**

The policy target should be \( \text{true or should evaluate to an error} \). For any \( \text{permit} \) rule \( r_j = \langle rb_j, \text{permit} \rangle \) (\( j < i \)) before rule \( r_i \), \( rb_j \) should not evaluate to \( \text{true} \) (i.e it should be \( \text{N/A or evaluates to an error} \)).

Hence, the reachability constraint is - policy target evaluates to \( \text{true or error} \) and all the previous rules with \( \text{permit} \) effect before the current rule under consideration should evaluate to \( \text{N/A or error} \) (i.e should not evaluate to \( \text{true} \)).

In formal notation, it will be \( \{PT \lor \text{Error}(PT)\} \land \{\neg rb_j \text{ for any rule } r_j \text{ for } j \neq i\} \lor \{\exists d (rb_d \lor \text{Error}(rb_d)) \land (r_d = \langle rb_d, \text{deny}\rangle) \land (\neg rb_j \lor \text{Error}(rb_j)) \text{ for any permit rule } r_j = \langle rb_d, \text{permit} \rangle \text{ for } j \neq i\})

ii) **Necessity constraint**

The necessity constraint for RPTE with \( i^{th} \) \( \text{AnyOf} \) element missing is that – rule target expression should have negation term for \( i^{th} \) \( \text{AnyOf} \) element and rule condition evaluates to \( \text{true} \).

Formally, \( (\text{AnyOf}_i \land \neg \text{AnyOf}_i \land \ldots \text{AnyOf}_n) \land rc_i \land r_i = \langle rb_i, \text{permit} \rangle \)

The necessity constraint for RPTE with \( j^{th} \) \( \text{AllOf} \) element of \( i^{th} \) \( \text{AnyOf} \) element missing is that – rule target expression should have a positive term for \( j^{th} \) \( \text{AllOf} \) element of \( i^{th} \) \( \text{AnyOf} \) element and all other \( \text{AllOf} \) element should have a negative term and rule condition evaluates to \( \text{true} \).
Formally, \((\text{Any}Of_i \land \text{Any}Of_i = (\neg \text{All}Of_i \lor \text{All}Of_j \lor \neg \text{All}Of_m) \land \ldots \land \text{Any}Of_n) \land rc_i\)

iii) **Propagation constraint**

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be deny, and, in original policy, it will be N/A or error. However, for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both of them will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect after the \(i^{th}\) rule should evaluate to N/A or error.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is *deny-unless-permit*.

\[
([PT \lor \text{Error}(PT)] \land \{(\text{Any}Of_i \land \neg \text{Any}Of_i \land \ldots \text{Any}Of_n) \lor (\text{Any}Of_i \land \text{Any}Of_i = (\neg \text{All}Of_i \\
\lor \text{All}Of_j \lor \neg \text{All}Of_m) \land \ldots \land \text{Any}Of_n)\} \land rc_i \land \{\neg rb \lor \text{Error}(rb) \text{ for any permit rule } r_j = rb, \text{ permit } > j \neq i\})
\]

c) **first-applicable:**

i) **Reachability constraint**

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to N/A.

ii) **Necessity constraint**

The necessity constraint for RPTE with \(i^{th}\) *AnyOf* element missing is that – rule target expression should have negation term for \(i^{th}\) *AnyOf* element and rule condition evaluates to *true*.

Formally, \((\text{Any}Of_i \land \neg \text{Any}Of_i \land \ldots \land \text{Any}Of_n) \land rc_i\)
The necessity constraint for RPTE with $j^{th}$ AllOf element of $i^{th}$ AnyOf element missing is that – rule target expression should have a positive term for $j^{th}$ AllOf element of $i^{th}$ AnyOf element and all other AllOf element should have a negative term and rule condition evaluates to $true$.

Formally, \((\text{AnyOf}_i \land \text{AnyOf}_i) = (\neg\text{AllOf}_i \lor \text{AllOf}_j \lor \neg\text{AllOf}_m) \land .. \text{AnyOf}_n) \land r_{ci}\)

iii) **Propagation constraint**

If policy target evaluates to $true$, then for all other rules which have the same effect as that of $i^{th}$ rule should evaluate to $N/A$ or error.

If policy target evaluates to an error, then for all other rules which have the same effect as that of $i^{th}$ rule should evaluate to $N/A$.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is first-applicable.

\[
(\{PT\} \land ((\text{AnyOf}_i \land \neg\text{AnyOf}_i \land .. \text{AnyOf}_n) \lor (\text{AnyOf}_i \land \text{AnyOf}_i) = (\neg\text{AllOf}_i \lor \text{AllOf}_j \lor \neg\text{AllOf}_m) \land .. \text{AnyOf}_n)) \land r_{ci} \land \{\neg rb_j for any rule r_j for j < i\} \land \{\neg rb_j \lor \text{Error}(rb_j) for any rule r_j such that re_i = re_j for j > i\})
\]

\[
\lor
\]

\[
(\text{Error}(PT)) \land ((\text{AnyOf}_i \land \neg\text{AnyOf}_i \land .. \text{AnyOf}_n) \lor (\text{AnyOf}_i \land \text{AnyOf}_i) = (\neg\text{AllOf}_i \lor \text{AllOf}_j \lor \neg\text{AllOf}_m) \land .. \text{AnyOf}_n)) \land r_{ci} \land \{\neg rb_j for any rule r_j for j < i\} \land \{\neg rb_j \lor \text{Error}(rb_j) for any rule r_j such that re_i = re_j for j > i\})
\]

3.14 **FDC for Change Rule Combining Algorithm (CRC)**

Change Rule Combining Algorithm is a mutation operator in which the combining algorithm of the policy is changed to another combining algorithm. Since it has a fault of in combining algorithm, it is under the category incorrect combining algorithm.
Consider an XACML policy \( P = <PT, RCA, RL> \) and \( P' = <PT, RCA', RL> \) such that \( P' \) is similar to \( P \) except the \( RCA' \) in \( P' \) is different from one in \( P \). Here, \( P' \) is called the Change Combining Algorithm (CCA) Mutant of \( P \). The fault detection condition for CRC presented as follows.

**a) FDC for change from permit-overrides to Deny-overrides:**

Policy target should evaluate to true or error.

There should exist at least one pair of rules such that one of them is permit rule and another is deny such that both evaluate to true or one evaluates to true and other evaluates to an error.

Formally,

\[
(PT \lor \text{Error}(PT)) \land \exists (p,d) \text{ such that } r_p = <rb_p, permit> \land r_d = <rb_d, deny> \land (rb_p \land rb_d) \lor \{\text{Error}(rb_p) \land rb_d\} \lor \{\text{Error}(rb_d) \land rb_p\})
\]

Note: if all of the permit rules are N/A while one or more deny rules evaluate to true or error and if all of the deny rules are N/A while one or more permit rules evaluate to true or error, permit-overrides and deny-overrides behaves the same.

**b) FDC for change from permit-overrides to deny-unless-permit:**

Policy target should evaluate to true or error.

(All of the rules evaluate to N/A

one or more permit rules evaluates to an error

one or more deny rules evaluates to an error and the rest are N/A and \( PT \) evaluates to true)
\[(PT \lor \text{Error}(PT)) \land (\{\neg rb\}_j \text{ for all rules in } P) \lor \{\text{Error}(rb) \text{ for any permit rule } r_j = <rb, permit> \in P\} \lor \{\text{PT} \land \text{Error}(rb_k) \text{ for some deny rule } r_k = <rb, deny> \in P \land \neg rb_j \text{ for other rules } r_j \in P \text{ such that } k \neq j}\]  

c) **FDC for change from permit-overrides to permit-unless-denies:**

Policy target should evaluate to `true` or `error`.

\[\land\]

(All of the rules evaluate to N/A)

\[\lor\]

one or more `permit` rule evaluates to `true` only if one or more `deny` rules evaluate to `true`.

\[\lor\]

one or more `deny` rule evaluates to `true` only if one or more `permit` rule evaluates to `true` or `error`.

\[\lor\]

one or more `permit` rule evaluates to an error and the rest evaluates to N/A and PT evaluates to `true`.)

Formally,

\[(PT \lor \text{Error}(PT)) \land (\{\neg rb\}_j \text{ for all rules in } P) \lor \{rb_d \land (rb \lor \text{Error}(rb)) \text{ for any permit rule } r_p = <rb, permit> \text{ and for any deny rule } r_d = <rb, deny> \in P \lor \{\text{PT} \land \text{Error}(rb_k) \text{ for some permit rule } r_k = <rb, permit> \in P \land \neg rb_j \text{ for other rules } r_j \in P \text{ such that } k \neq j}\]  

d) **FDC for change from permit-overrides to first-applicable:**

Policy target should evaluate to `true` or `error`.

\[\land\]
if \( r_f \) is the first rule that evaluates to true, then \( r_f \) is deny rule and there exists one or more permit rule that evaluates to true or error

\( \lor \)

if \( r_f \) is the first rule that evaluates to an error and \( r_f \) is deny rule, then there exists one or more permit or deny rule that evaluates to true or one or more permit rule that evaluates to an error

\( \lor \)

if \( r_f \) is the first rule that evaluates to an error and \( r_f \) is permit rule, then there exists one or more permit or deny rule that evaluates to true or one or more deny rule that evaluates to an error

Formally,

\[ (PT \lor \text{Error}(PT)) \]
\[ \land \]
\[ \{ (\neg rb_j \land rb_f \text{ for some } f \text{ such that } j < f) \land (r_f = \langle rb_f, \text{ deny} \rangle \land rb_p \text{ for some permit rule } r_p = \langle rb_p, \text{ permit} \rangle) \} \]
\[ \lor \]
\[ \{ (\neg rb_j \land \text{Error}(rb_f) \text{ for some } f \text{ such that } j < f) \land (r_f = \langle rb_f, \text{ deny} \rangle) \land (rb_i \text{ for some permit or deny rule } \lor \text{Error}(rb_p) \text{ for some permit rule } rb_p) \} \}
\[ \lor \]
\[ \{ (\neg rb_j \land \text{Error}(rb_f) \text{ for some } f \text{ such that } j < f) \land (r_f = \langle rb_f, \text{ permit} \rangle) \land (rb_i \text{ for some permit or deny rule } \lor \text{Error}(rb_d) \text{ for some deny rule } rb_d) \} \}

e) FDC for change from deny-overrides to permit-unless-deny:

Policy target should evaluate to true or error.
(All of the rules evaluate to N/A

\lor

one or more deny rule evaluates to an error

\lor

one or more permit rule evaluates to an error and rest are N/A and PT evaluates to true)

Formally,

\((PT \lor Error(PT)) \land \{\neg rb_j\ \text{for all rules in } P\} \lor \{Error(rb_j) \\text{for any deny rule } r_j = <rb_j, deny>\ \text{in } P\} \lor \{PT \land Error(rb_k) \\text{for some permit rule } r_k = <rb_k, permit>\ \text{in } P \land \neg rb_j\ \text{for other rules } r_j \ \text{in } P \text{ such that } k \neq j\})\)

f) **FDC for change from deny-overrides to deny-unless-permit:**

Policy target should evaluate to true or error.

\land

(All of the rules evaluate to N/A

\lor

one or more deny rule evaluates to true only if one or more permit rules evaluate to true.

\lor

one or more permit rule evaluates to true only if one or more deny rule evaluates to true or error

\lor

one or more deny rule evaluates to an error and rest evaluates to N/A and PT evaluates to true.)

Formally,
(PT ∨ Error(PT)) ∧ (\{¬rb \text{ for all rules in } P \} ∨ \{ rb_p ∧ (rb_d ∨ Error(rb_d)) \}) for any permit rule \( r_p = <rb_p, \text{ permit}> \) and for any deny rule \( r_d = <rb_d, \text{ deny}> \) in \( P \) \( ∨ \{PT ∧ Error(rb_k) \) for some deny rule \( r_k = <rb_k, \text{ deny}> \) in \( P \) \( ∧ ¬rb \) for other rules \( r_j \) in \( P \) such that \( k \neq j \})

\textbf{g) FDC for change from \textit{deny-overrides} to \textit{first-applicable}:}

Policy target should evaluate to \textit{true} or \textit{error}.

∧

if \( r_f \) is the first rule that evaluates to \textit{true}, then \( r_f \) is \textit{permit} rule and there exists one or more \textit{deny} rule that evaluates to \textit{true} or \textit{error}

∨

if \( r_f \) is the first rule that evaluates to an error and \( r_f \) is \textit{permit} rule, then there exists one or more \textit{permit} or \textit{deny} rule that evaluates to \textit{true} or one or more \textit{deny} rule that evaluates to an error

∨

if \( r_f \) is the first rule that evaluates to an error and \( r_f \) is \textit{deny} rule, then there exists one or more \textit{permit} or \textit{deny} rule that evaluates to \textit{true} or one or more \textit{permit} rule that evaluates to an error

Formally,

\[(PT ∨ Error(PT)) \land ((¬rb_f \land rb_f \text{ for some } f \text{ such that } j < f) \land (r_f = <rb_f, \text{ permit}> \land rb_d \text{ for some } \text{ deny } \text{ rule } r_d = <rb_d, \text{ deny}>)) \]
\[ \{(\neg rb_j \land Error(rb_f) \text{ for some } f \text{ such that } j < f) \land (r_f = <rb_f, permit>) \land (rb_i \text{ for some permit or deny rule} \lor Error(rb_d) \text{ for some deny rule } rb_d) )\} \lor \]

\[ \{(\neg rb_j \land Error(rb_f) \text{ for some } f \text{ such that } j < f) \land (r_f = <rb_f, deny>) \land (rb_i \text{ for some deny or permit rule} \lor Error(rb_p) \text{ for some permit rule } rb_p) )\} \]

h) **FDC for change from permit-unless-deny to deny-unless-permit:**

Policy target should evaluate to true or error.

\[ \land \]

(All of the rules evaluate to N/A or error

\[ \lor \]

if one rule evaluates to true, then other rules with opposite effect must evaluate to true.)

Formally,

\[ (PT \lor Error(PT)) \land \{(\neg rb_j \lor Error(rb_j) \text{ for all rules in } P) \lor \{ rb_p \land rb_d \text{ for any permit rule } r_p = <rb_p, permit> \text{ and for any deny rule } r_d = <rb_d, deny> \text{ in } P}) \]

i) **FDC for change from permit-unless-deny to first-applicable:**

Policy target should evaluate to true or error

\[ \land \]

if \( r_f \) is the first rule that evaluates to true, then \( r_f \) is permit rule and there exists one or more deny rule that evaluates to true.

\[ \lor \]

if \( r_f \) is the first rule that evaluates to an error and \( r_f \) is permit rule, then policy target should evaluate to true or one or more deny rule should evaluate to true.
if $r_f$ is the first rule that evaluates to an error and $r_f$ is *deny* rule, then policy target should evaluate to *true* if one or more *deny* rule evaluates to *true.*

Formally,

$$(PT \lor \text{Error}(PT))$$

\&

{$$((\neg rb_j \land rb_f \text{ for some } f \text{ such that } j < f) \land (r_f = <rb_f, \text{ permit}> \land rb_d \text{ for some deny rule } r_d = <rb_d, \text{ deny}>))$$}

\lor

{$$((\neg rb_j \land \text{Error}(rb_f) \text{ for some } f \text{ such that } j < f) \land (r_f = <rb_f, \text{ permit}> \land (rb_d \text{ for some deny rule } r_d = <rb_d, \text{ deny}>)) \lor PT)$$}

\lor

{$$((\neg rb_j \land \text{Error}(rb_f) \text{ for some } f \text{ such that } j < f) \land (r_f = <rb_f, \text{ deny}> \land (\neg (rb_d \text{ for some deny rule } r_d = <rb_d, \text{ deny}>)) \lor PT)$$}

j) **FDC for change from *deny-unless-permit* to *first-applicable***:

Policy target should evaluate to *true* or *error*

\&

if $r_f$ is the first rule that evaluates to *true*, then $r_f$ is *deny* rule and there exists one or more *permit* rule that evaluates to *true*.

\lor

if $r_f$ is the first rule that evaluates to an error and $r_f$ is *deny* rule, then policy target should evaluate to *true* or one or more *permit* rule should evaluate to *true.*
if \( r_f \) is the first rule that evaluates to an error and \( r_f \) is permit rule, then policy target should evaluate to \textit{true} if one or more permit rule evaluates to \textit{true}.)

Formally,

\[
(PT \lor \text{Error}(PT))
\]

\[
\land
\]

\[
\{(\neg rb_j \land rb_f \text{ for some } f \text{ such that } j < f) \land (r_f = \langle rb_p, \text{deny} \rangle \land rb_p \text{ for some permit rule } r_p = \langle rb_p, \text{permit} \rangle)\}
\]

\[
\lor
\]

\[
\{(\neg rb_j \land \text{Error}(rb_f) \text{ for some } f \text{ such that } j < f) \land (r_f = \langle rb_p, \text{deny} \rangle \land (rb_p \text{ for some permit rule } r_p = \langle rb_p, \text{permit} \rangle) \lor PT)\}
\]

\[
\lor
\]

\[
\{(\neg rb_j \land \text{Error}(rb_f) \text{ for some } f \text{ such that } j < f) \land (r_f = \langle rb_p, \text{permit} \rangle \land (\neg (rb_p \text{ for some permit rule } r_p = \langle rb_p, \text{permit} \rangle)) \lor PT)\}
\]

\textbf{Note:} FDC for CRC from permit-overrides to deny-overrides is applicable from deny-overrides to permit-overrides as well. Hence, we have all possible combinations between five RCAs.
CHAPTER 4

Mutation-Based Test Generation

Mutation-based test generation involves generating test suites exploiting the fault detection condition with the goal to kill all the non-equivalent mutants. The formalization of complete fault detection conditions makes it feasible to automatically perform strong mutation-based test generation. From fault detection condition, we know that if any request satisfies reachability, necessity and propagation for a fault, it will produce a different result in original policy, and faulty policy. This difference in response can be exploited to distinguish faulty policy from the correct policy. Hence, the goal is to generate test input that satisfies the three constraints of the fault detection condition - reachability, necessity and sufficiency/propagation for each possible fault policy.

FDC may include multiple mutually exclusive conditions. If one of them is satisfied, it will be sufficient for the fault detection. As a result, we just use one of the mutually exclusive conditions for each fault. In other words, we do not need to generate multiple requests for each of the mutually exclusive conditions because it will just generate redundant test cases. The process of identifying such sufficient mutually exclusive conditions (if any exist) to detect a fault is discussed in Section 1.3.2. The overall process for strong mutation-based (SMT) test generation is depicted in Algorithm 1.
4.1 Strong Mutation-Based Test Generation

Algorithm 1: Mutation-based generation of the near-optimal test suite (SMT)

*Import functions*: \( \text{kill}(M, Q) \) returns the list of mutants in \( M \) that are killed by test suite \( Q \)

\[ Z3\text{-request}(FDC) \] returns a solution to the constraint \( FDC \)

*Input*: Policy \( P = \langle PT, RCA, [r_1, r_2, ..., r_n]\rangle \), \( \Omega \) is a list of mutation operators

*Output*: A set of access requests \( Q \)

*Variables*: \( M \) is a mutant pool, \( OPS \) is a list of mutation operators, \( FDC \) is a fault detection condition, \( q \) is a test input

1. \( Q \leftarrow \emptyset \)
2. While \( \Omega \neq \emptyset \)
3. \( OPS \leftarrow \text{select one or more mutation operators from } \Omega \)
4. \( M \leftarrow \text{list of mutants created by mutation operators } OPS \)
5. \( M \leftarrow M - \text{kill}(M, Q) \)
6. While \( M \neq \emptyset \)
7. \( FDC \leftarrow \text{compose the fault detection conditions of one or more mutants} \)
8. \( q \leftarrow Z3\text{-request}(FDC) \)
9. If \((q \text{ is not null}) \) // otherwise equivalent mutant
10. \( Q \leftarrow Q \cup \{q\} \)
11. \( M \leftarrow M - \text{kill}(M, \{q\}) \)
12. EndIf
13. EndWhile
14. EndWhile
15. Return \( Q \)
From formalization of the fault detection condition, we have identified RTF, RCF, ANF, RNF, and RER have common fault detection conditions among various possible mutually exclusive fault detection conditions for each of them. Algorithm 6 in Section 4.3 presents such common fault detection conditions among them. As a result, we do not need to deal with each of those mutation operators individually. In other words, if we generate a request that satisfies that common fault detection condition among them, it can kill the mutants from RTF, RCF, ANF, RNF and RER. So, we select one (or more mutation operators if they have common FDC ) at a time and generate mutants for them (line 3-4), and run the mutants against the existing tests. The mutants that are already killed by the existing tests are removed from the mutant pool (line 5). Then we compose the constraint for one or more compatible mutants (line 7) and then solve the constraint by using the Z3 constraint solver [17] (line 8) If the constraint is solved, we convert the solution into an access request and add it to the test suite (lines 9-10), otherwise, the mutant is considered an equivalent mutant. We also run the new test against the current mutant pool. Mutants killed by the new test will be removed from the pool (line 11). When all mutation operators are handled, we return Q as the generated test suite.

Algorithm 1 is computationally expensive for the policy with a large number of the rules because of the involved optimization (step 4, 5 and 11) which needs to generate mutants and determine whether they are killed or not. These optimization steps are costly because \( \text{kill}(M,Q) \) is in the order of \( O(n^3) \) where \( n \) is the number of the rule in a policy. Executing a test with each mutant has a complexity of \( O(n) \). The number of mutants is in linear order and test suite size is linear as well. Hence, \( \text{kill}(M,Q) \) is in the order of \( O(n^3) \).
because in the worst case we need to run an operation with the complexity of \( O(n) \) for mutants of the size \( O(n) \) for each request in test suite of the size \( O(n) \).

As a result, if \( n \) grows very large in big policies, the optimization steps grows in the cubic order which makes it unfeasible to use for large policies. So, we also formulated the mutation-based test generation without optimization which we referred to as Non-Optimized Strong Mutation-based Testing (NO-SMT) which generates a test suite that achieves a 100% mutant score like SMT. NO-SMT is nothing but the Algorithm 1 itself excluding the steps 4, 5 and 11. However, it will have many redundant test cases than that from SMT. Hence, it is the trade-off between test generation time against test suite size.

We have implemented these algorithms as an extension to the open source XPA tool.

The complexity of step 7 (for a single iteration) i.e generation of FDC constraint will be in the order of \( O(n) \). The details on FDC constraint generation for each mutation operator is presented in Section 4.1 through Section 4.14. The time complexity of step 8 (for single iteration) is the time complexity of Z3 for solving FDC constraint expression. Let, \( O(Z3) \) represents the time complexity of Z3 at step 8. \( \text{kill}(M, \{q\}) \) in step 11 will be in the order of \( O(n^2) \) since it is similar to \( \text{kill}(M,Q) \) but for a single test \( q \). Since the size of \( M \) will be in linear order, the time complexity of step 6 through step 13 will be in the order of \( O(n.(O(Z3)+ n^2)) \).

As demonstrated in the empirical study, the test generation time of SMT grows much faster than \( RC, DC, \) and \( MC/DC \). It is worth pointing out that, in this paper, the mutation-based generation of the near-optimal test suite is not meant to be a practical test generation method for large-scale XACML policies. Instead, our goal is to use the test suite as a
benchmark for measuring relative cost-effectiveness of other testing methods. In addition, the mutation-based test suite can be used to help improve other testing methods by determining equivalent mutants and investigating fault detection conditions of live mutants. However, coverage-based test generation does not guarantee 100% mutation score and if 100% mutation score is desired irrespective of the size of a test suite, then we can use NO-SMT for the larger policies as well.

The following are algorithms used to construct fault detection condition for each possible mutant for each mutation operator in the fault model.

4.1.1 FDC Constraint and Tests Generation for CRE

Algorithm 2: generateTestsForCRE(P)

Input: Policy P = < PT, RCA, [r₁, r₂, ..., rₙ] >

Output: A set of access requests Q

1. \( Q \leftarrow \emptyset \)
2. \( \text{constraint} \leftarrow PT \)
3. For each Rule \( r_i \) in \( [r₁, r₂, ..., rₙ] \) in \( P \), do
4. \( \text{ruleConstraint} \leftarrow \text{constraint} \)
5. \( \text{ruleConstraint} \leftarrow \text{ruleConstraint} \land \text{ruleReachability}(P, r_i) \)
6. \( \text{ruleConstraint} \leftarrow \text{ruleConstraint} \land \text{rt}_i \land \text{rc}_i \)
7. \( \text{ruleConstraint} \leftarrow \text{ruleConstraint} \land \text{rulePropagation}(P, r_i, CRE) \)
8. \( Q \leftarrow Q \cup \text{Z3-request}(\text{ruleConstraint}) \)
9. EndFor
10. return \( Q \)
The input to the algorithm is policy $P = < PT, RCA, [r_1, r_2, ..., r_n]>$ and output is set of generated access request $Q$. Initially we set $Q$ to be empty (line 1). We then set the constraint expression from policy target if it is present, otherwise, the constraint is empty (line 4). The next step is to iterate over each rule in the policy (line 3) and construct fault detection condition constraint to kill the mutant. For each rule, the rule constraint is set to constraint expression (line 4) from policy target constraint in line 2. Then, for each rule in the policy, the rule constraint expression is concatenated with reachability constraint (line 5). Once the reachability constraint is met, the next task is to concatenate the rule constraint with necessity constraint for the fault type (line 6) and then concatenate the propagation constraint (line 7). The constructed rule constraint is the constraint with sufficient constraint to kill the CRE mutant and is supplied to the constraint solver (line 8) to obtain the value for the test inputs. When all the rules are processed, the generated set of test input - $Q$ is returned. The algorithm for rule reachability is presented in Algorithm 3 in Section 4.1.1 and that for rule propagation is presented in Algorithm 4 in Section 4.1.2. It should be noted that reachability and propagation constraint is constructed taking common mutually exclusive conditions for reachability and propagation constraint for RTT, RTF, RCT, RCF, ANF, RNF and RER such that it works for all of them. CRE has slightly different propagation constraint than others but still shares the common constraints among them. Hence, rather than defining redundant propagation constraint for CRE only, we used the same method for propagation constraint passing the mutation method itself as an argument to tweak the propagation constraint based on the mutation operator.
4.1.2 FDC Constraint for Rule Reachability

Algorithm 3: ruleReachability(P, r_i)

Import functions: hasCommonAttribute(rb_i, rb_j) \( \Rightarrow \) returns true if rule body of \( i^{th} \) rule and rule body of \( j^{th} \) rule has a common attribute

Input: Policy \( P = < PT, RCA, [r_1, r_2, ..., r_n] > \), Rule \( r_i = < rt_i, rc_i, re_i > \)

Output: constraint

1. \( \text{constraint} \leftarrow \epsilon \)

2. If RCA = First-Applicable, then

3. For each rule \( r_k \in [r_1, r_2, ..., r_{i-1}] \) in P, do

4. \( \text{constraint} \leftarrow \text{constraint} \land \neg (rt_k \land rc_k) \)

5. endFor

6. Else

7. if RCA = Permit-unless-Deny, then

8. for each rule \( r_k \in [r_1, r_2, ..., r_{i-1}] \) in P, do

9. if \( re_k = \text{Deny} \), then

10. \( \text{constraint} \leftarrow \text{constraint} \land \neg (rt_k \land rc_k) \)

11. endIf

12. endFor

13. Else if RCA = Deny-unless-Permit, then

14. for each rule \( r_k \in [r_1, r_2, ..., r_{i-1}] \) in P, do

15. if \( re_k = \text{Permit} \), then

16. \( \text{constraint} \leftarrow \text{constraint} \land \neg (rt_k \land rc_k) \)

17. endIf
else if \( RCA = \text{Deny-overrides} \), then

for each rule \( r_k \) in \( [r_1, r_2, \ldots, r_{i-1}] \) in \( P \), do

if \( r_k = \text{Deny} \), then

if hasCommonAttributes(\( rb_i, rb_k \), then

\[ \text{constraint} \leftarrow \text{constraint} \land \neg (rt_k \land rc_k) \]

else

\[ \text{dominantRuleCollection.add}(r_k) \]

endIf

endIf

endFor

else if \( RCA = \text{Permit-overrides} \), then

for each rule \( r_k \) in \( [r_1, r_2, \ldots, r_{i-1}] \) in \( P \), do

if \( r_k = \text{Permit} \), then

if hasCommonAttributes(\( rb_i, rb_k \), then

\[ \text{constraint} \leftarrow \text{constraint} \land \neg (rt_k \land rc_k) \]

else

\[ \text{dominantRuleCollection.add}(r_k) \]

endIf

endIf

endFor

endIf
4.2 Return Constraint

If rule combining algorithm is first-applicable, then all the previous rules before the current rule under consideration i.e. \(i^{th}\) rule should evaluate to N/A (line 2-5) otherwise if RCA is permit-unless-deny, then all the previous rules with deny effect should be N/A (line 7-12). If RCA is deny-unless-permit, all the previous rules with permit effect should be N/A (line 13-18). If RCA is deny-overrides (or permit-overrides), then we only falsify deny (or permit) rules with common attributes in rule body of current rule under consideration and mark the deny (or permit) rules with no common attribute for later to be used for propagation constraint (line 19-29 and line 30-41).

4.3 FDC Constraint for Propagation

**Algorithm 4: rulePropagation(\(P, r_i, mutationMethod\))**

*Input: Policy \(P = < PT, RCA, [r_1, r_2, ..., r_n]>\), Rule \(r_i = <rt_i, rc_i, re_i>\)*

*Output: constraint*

1. \(\text{constraint} \leftarrow e\)

2. If RCA = First-Applicable, then

   3. for each rule \(r_k\) in \([r_{i+1}, r_{i+2}, ..., r_n]\) in \(P\), do

   4. if \(re_k = re_i\), then

   5. \(\text{constraint} \leftarrow \text{constraint} \land \neg (rt_k \land rc_k)\)

   6. endIf

   7. endFor

8. else

9. if RCA = Permit-unless-Deny, then

10. for each rule \(r_k\) in \([r_{i+1}, r_{i+2}, ..., r_n]\) in \(P\), do
if \( r_e \) = Deny, then

\[
\text{constraint} \leftarrow \text{constraint} \land \neg (r_t \land r_c)
\]

endIf

endFor

else if R\( \text{CA} \) = Deny-unless-Permit, then

for each rule \( r_k \) in \( [r_{i+1}, r_{i+2}, \ldots, r_n] \) in \( P \), do

if \( r_e \) = Permit, then

\[
\text{constraint} \leftarrow \text{constraint} \land \neg (r_t \land r_c)
\]

endIf

endIf

else if R\( \text{CA} \) = Deny-overrides, then

for each rule \( r_k \) in \( [r_{i+1}, r_{i+2}, \ldots, r_n] \) in \( P \), do

if \( r_e \) = Deny, then

if hasCommonAttributes(rb\( _i \), rb\( _k \)), then

\[
\text{constraint} \leftarrow \text{constraint} \land \neg (r_t \land r_c)
\]

else

dominantRuleCollection.add(r\( _k \))

endIf

endIf

endFor

if \( r_{e1} \) = permit and mutationMethod \( \neq \) CRE then

for each rule \( r_l \) in dominantRuleCollection, do

if \( \neg (r_t \land r_c) \rightarrow (r_t \land r_c) \) for \( r_p = <rb_p, \text{permit}> \), then
\[
\text{dominantIndeterminateFlag} \leftarrow \text{true}
\]

\[
\text{else}
\]

\[
\text{constraint} \leftarrow \text{constraint} \land \neg (r_{t_i} \land r_{c_i})
\]

\[
\text{endIf}
\]

\[
\text{endFor}
\]

\[
\text{if dominantIndeterminateFlag = true, then}
\]

\[
\text{permit-rules} \leftarrow \text{get permit rules of P}
\]

\[
\text{for each } r_p \text{ in permit-rules, do}
\]

\[
\text{constraint} \leftarrow \text{constraint} \land (r_{t_p} \land r_{c_p})
\]

\[
\text{endFor}
\]

\[
\text{endIf}
\]

\[
\text{endIf}
\]

\[
\text{else if } \text{RCA} = \text{permit-overrides}, \text{ then}
\]

\[
\text{for each rule } r_k \text{ in } [r_{i+1}, r_{i+2}, \ldots, r_n] \text{ in } P, \text{ do}
\]

\[
\text{if } r_{e_k} = \text{permit}, \text{ then}
\]

\[
\text{if hasCommonAttributes}(r_b, r_{b_k}), \text{ then}
\]

\[
\text{constraint} \leftarrow \text{constraint} \land \neg (r_{t_k} \land r_{c_k})
\]

\[
\text{else}
\]

\[
\text{dominantRuleCollection.add}(r_k)
\]

\[
\text{endIf}
\]

\[
\text{endIf}
\]

\[
\text{endFor}
\]
if \( r_{ei} = \text{deny} \) and \( \text{mutationMethod} \neq \text{CRE} \) then

for each rule \( r_i \) in \( \text{dominantRuleCollection} \), do

if \( \neg (r_{ti} \land r_{ci}) \rightarrow (r_{tp} \land r_{cp}) \) for \( r_p = <r_{bp}, \text{permit}> \), then

\[ \text{dominantIndeterminateFlag} \leftarrow \text{true} \]

else

\[ \text{constraint} \leftarrow \text{constraint} \land \neg (r_{ti} \land r_{ci}) \]

endIf

endFor

if \( \text{dominantIndeterminateFlag} = \text{true} \), then

\[ \text{deny-rules} \leftarrow \text{get deny rules of P} \]

for each \( r_d \) in \( \text{deny-rules} \), do

\[ \text{constraint} \leftarrow \text{constraint} \land (r_{td} \land r_{cd}) \]

endFor

endIf

endIf

endIf

endIf

return \( \text{constraint} \)

If RCA is first-applicable, then all the rules after current rule under consideration with the same effect as that of the current rule under consideration should evaluate to N/A. If RCA is permit-unless-deny, then all the rules after \( i^{th} \) rule with deny effect should be N/A. If RCA is deny-unless-permit, all the rules after \( i^{th} \) rule with permit effect should be N/A. If RCA is deny-overrides (or permit-overrides), then we only falsify deny (or permit) rules
with common attributes in rule body of current rule under consideration and mark the deny
(or permit) rules with no common attribute for later to be used for propagation constraint.
If \(i^{th}\) rule is deny (or permit), then propagation constraint is satisfied, and we can return the
constraint otherwise if it is permit (or deny) and mutation operator is not CRE (for CRE
the later propagation constraint i.e step 31-46 is not required) and \(\text{dominantRuleCollection}\)
is not empty, we need to check for each rule in \(\text{dominantRuleCollection}\) whether falsifying
this deny (or permit) rule will fire another permit (or deny) rule. If this is the case, we
cannot falsify the current rule in \(\text{dominantRuleCollection}\). Hence, we make it evaluate
indeterminate and set \(\text{dominantIndeterminateFlag}\) to true. If any deny (or permit) rule
evaluate to indeterminate, then for fault detection, all the permit(deny) rules should
evaluate to \(N/A\).

4.4 FDC Constraint and Tests Generation for RTT

Algorithm 5: \(\text{generateTestsForRTT}(P)\)

\(\text{Input: Policy } P = \langle PT, RCA, \{r_1, r_2, \ldots, r_n\}\rangle\)

\(\text{Output: A set of access requests } Q\)

1 \(Q \leftarrow c\)

2 \(\text{dominantIndeterminateFlag } \leftarrow \text{false}\)

3 \(\text{dominantRuleCollection } \leftarrow \text{null}\)

4 \(\text{constraint } \leftarrow PT\)

5 \(\text{For each Rule } r_i \text{ in } \{r_1, r_2, \ldots, r_n\} \text{ in } P, \text{ do}\)

6 \(\text{ruleConstraint } \leftarrow \text{constraint}\)

7 \(\text{ruleConstraint } \leftarrow \text{ruleConstraint } \land \text{ruleReachability}(P, r_i)\)

8 \(\text{ruleConstraint } \leftarrow \text{ruleConstraint } \land \lnot r_i \land (r_i \lor \text{Error}(r_i))\)
Like CRE, we construct the FDC to kill the RTT by combining reachability, necessity and propagation constraint. The necessity constraint for RTT is that target of the rule under consideration should be N/A and rule condition should be true.

4.5 FDC constraint and tests generation for RTF

Algorithm 6: generateTestsForRTF(P)

Input: Policy $P= \langle PT, RCA, [r_1, r_2, \ldots, r_n] \rangle$

Output: A set of access requests $Q$

1. $Q \leftarrow \emptyset$
2. $dominantIndeterminateFlag \leftarrow false$
3. $dominantRuleCollection \leftarrow null$
4. $constraint \leftarrow PT$
5. For each Rule $r_i$ in $[r_1, r_2, \ldots, r_n]$ in $P$, do
6. \hspace{1cm} $ruleConstraint \leftarrow constraint$
7. \hspace{1cm} $ruleConstraint \leftarrow ruleConstraint \land ruleReachability(P, r_i)$
8. \hspace{1cm} $ruleConstraint \leftarrow ruleConstraint \land (rt_i \land rc_i) \lor Error(rb_i)$
9. \hspace{1cm} $ruleConstraint \leftarrow ruleConstraint \land rulePropagation(P, r_i, RTF)$
10. \hspace{1cm} $Q \leftarrow Q \cup Z3-request(ruleConstraint)$
11. EndFor
12. return $Q$
The input to the algorithm is policy \( P = < PT, RCA, [r_1, r_2, ..., r_n] > \) and output is set of generated access request \( Q \). Initially we set \( Q \) to be empty (line 1). We set the `dominantIndeterminateFlag` to `false` and `dominantRuleCollection` to `null` to keep track of whether any dominant rule evaluated to indeterminate or not. If \( RCA \) is `first-applicable`, then any rule is dominant. If \( RCA \) is `permit-overrides` or `deny-unless-permit`, then the rules with permit effect will be the dominant rule. Similarly, rules with `deny` effect will be dominant in policy with `deny-overrides` or `permit-unless-deny` \( RCA \). We then set the constraint expression from policy target if it is present, otherwise, the constraint is empty (line 4). The next step is to iterate over each rule in the policy (line 5) and construct sufficient condition constraint to kill the mutant. For each rule, the rule constraint is set to constraint expression (line 6) from policy target constraint in line 2. Then, for each rule in the policy, the rule constraint expression is concatenated with reachability constraint (line 7). Once the reachability constraint is met, the next task is to concatenate the rule constraint with necessity constraint for the fault type (line 8) and then concatenate the propagation constraint (line 9). The constructed rule constraint is the constraint with sufficient condition to kill the RTF mutant which is supplied to constraint solver (line 10) to obtain the value for the test inputs. When all the rules are processed, the generated set of test input - \( Q \) is returned.

4.6 FDC Constraint and Test Generation for RCT

**Algorithm 7: generateTestsForRCT(P)**

*Input: Policy \( P = < PT, RCA, [r_1, r_2, ..., r_n] > \)*

*Output: A set of access requests \( Q \)*

1. \( Q \leftarrow \emptyset \)
constraint $\leftarrow PT$

For each Rule $r_i$ in $[r_1, r_2, ..., r_n]$ in $P$, do

\[ \text{ruleConstraint} \leftarrow \epsilon \]

\[ \text{ruleConstraint} \leftarrow \text{constraint} \land \text{ruleReachability}(P, r_i) \]

\[ \text{ruleConstraint} \leftarrow \text{ruleConstraint} \land (rt_i \lor \text{Error}(rt_i)) \land \neg rc_i \]

\[ \text{ruleConstraint} \leftarrow \text{ruleConstraint} \land \text{rulePropagation}(P, r_i, RCT) \]

\[ Q \leftarrow Q \cup Z3\text{-request}(\text{ruleConstraint}) \]

endFor

return $Q$

The sufficient condition to kill the RCT mutants is same as that for CRE except for that condition of the rule under consideration should be N/A.

4.7 FDC Constraint and Test Generation for RCF

RCF and RTF have common sufficient condition for fault detection. Hence, Algorithm 6 for RTF is enough, and we don’t need to consider RCF if RTF is in the fault model.

4.8 FDC Constraint and Test Generation for ANF

RCF and RTF have common sufficient condition for fault detection. Hence, Algorithm 6 for CRE is enough and we don’t need to consider ANF if CRE is in the fault model.

4.9 FDC Constraint and Test Generation for RNF

RNF and RTF have similar sufficient condition to kill them. Hence, the constraint for RTF is enough and we don’t need to consider RNF if RTF is in the fault model.

4.10 FDC Constraint and Test Generation for RER

RER and RTF have similar sufficient condition to kill them. Hence, the constraint for RTF is enough and we don’t need to consider RER if RTF is in the fault model.
4.11 FDC Constraint and Test Generation for PTT

**Algorithm 10**: generateTestsForPTT(P)

*Input*: Policy P= < PT, RCA, [r₁, r₂, ..., rₙ] >

*Output*: A set of access requests Q

1. \( Q \leftarrow \emptyset \)
2. \( \text{constraint} \leftarrow \neg PT \land (rt₁ \land rc₁) \)
3. \( Q \leftarrow Q \cup Z₃\text{-request(constraint)} \)
4. Return Q

The sufficient condition to kill PTT mutant is to make policy target evaluate to N/A and make any rule’s (say the first rule) target and condition evaluate to true.

4.12 FDC Constraint and Test Generation for PTF

**Algorithm 11**: generateTestsForPTF(P)

*Input*: Policy P= < PT, RCA, [r₁, r₂, ..., rₙ] >

*Output*: A set of access requests Q

1. \( Q \leftarrow \emptyset \)
2. \( \text{constraint} \leftarrow PT \land (rt₁ \land rc₁) \)
3. \( Q \leftarrow Q \cup Z₃\text{-request(constraint)} \)
4. Return Q

The sufficient condition to kill PTF mutant is to make policy target evaluate to true and make any rule’s (say the first rule) target and condition evaluate to true.

4.13 FDC Constraint and Test Generation for FPR

**Algorithm 8**: generateTestsForFPR(P)
Input: Policy \( P = \langle PT, RCA, \{r_1, r_2, \ldots, r_n\} \rangle \)

Output: A set of access requests \( Q \)

1. \( Q \leftarrow \emptyset \)
2. \( constraint \leftarrow PT \)
3. \( d \leftarrow \text{find the position of first Deny Rule} \)
4. \( p \leftarrow \text{find first Permit Rule after } d \)
5. \( \text{if such } p \text{ and } d \text{ does not exist} \)
6. \( \quad \text{return} \)
7. \( constraint \leftarrow rt_d \land rc_d \)
8. \( constraint \leftarrow rt_p \land rc_p \)
9. \( \text{if } RCA = \text{first-applicable} \)
10. \( \quad \text{For each rule } r_i \text{ in } \{r_1, r_2, \ldots, r_{d-1}\} \text{ in } P, \text{ do} \)
11. \( \quad \quad constraint \leftarrow constraint \land \neg (rt_i \land rc_i) \)
12. \( Q \leftarrow Q \cup Z3\text{-request}(constraint) \)
13. \( \text{return } Q \)

The FPR mutants behave similarly to the original policy if RCA is other than first-applicable. Even for the first applicable, if there does not exist a pair of rules such that one is deny and other is permit, then the FPR mutant behaves the same as original policy. Hence, the sufficient condition to kill non-equivalent FPR mutant is to make target and condition of a pair of rules evaluate to \text{true} such that one of them is permit rule and the other is deny rule.

4.14 FDC constraint and test generation for FDR

Algorithm 9: generateTestsForFDR(P)
Input: Policy \( P = \langle PT, RCA, [r_1, r_2, ..., r_n] \rangle \)

Output: A set of access requests \( Q \)

1. \( Q \leftarrow \emptyset \)
2. \( \text{constraint} \leftarrow PT \)
3. \( p \leftarrow \text{find the position of first permit Rule} \)
4. \( d \leftarrow \text{find first deny Rule after } p \)
5. if such \( p \) and \( d \) does not exist
   6. \( \text{return} \)
7. \( \text{constraint} \leftarrow rt_d \land rc_d \)
8. \( \text{constraint} \leftarrow rt_p \land rc_p \)
9. if \( RCA = \text{first-applicable} \)
10. \( \text{For each rule } r_i \text{ in } [r_1, r_2, ..., r_{d-1}] \text{ in } P, \text{ do} \)
11. \( \text{constraint} \leftarrow \text{constraint} \land \neg (rt_i \land rc_i) \)
12. \( Q \leftarrow Q \cup \text{request(constraint)} \)
13. \( \text{return } Q \)

The FDR mutants behave similarly to the original policy if \( RCA \) is other than \( \text{first-applicable} \). Even for the first applicable, if there does not exist a pair of rules such that one is deny and other is permit, then the FDR mutant behaves the same as original policy.

Hence, the sufficient condition to kill non-equivalent FDR mutant is to make target and condition of a pair of the rules evaluate to \( \text{true} \) such that one of them is permit rule and the other is deny rule.

4.15 FDC Constraint and Test Generation for RPTE

Algorithm 12: generateTestsForRPTE(\( P \))
Input: Policy $P = < PT, RCA, [r_1, r_2, ..., r_n] >$

Output: A set of access requests $Q$

1. $Q \leftarrow \emptyset$
2. $constraint \leftarrow PT$
3. For each Rule $r_i$ in $[r_1, r_2, ..., r_n]$ in $P$, do
   4. $ruleConstraint \leftarrow constraint \land ruleReachability(P, r_i)$
   5. $ruleConstraint \leftarrow ruleConstraint \land (rc_i \lor Error(rc_i))$
   6. $ruleConstraint \leftarrow ruleConstraint \land rulePropagation(P, r_i)$
   7. For each $\text{AnyOf}_j$ in $[\text{AnyOf}_{i_1}, \text{AnyOf}_{i_j}, ..., \text{AnyOf}_{i_m}]$ in $rt_i$, do
      8. $anyConstraint \leftarrow ruleConstraint \land \text{AnyOf}_{i_1} \land \neg \text{AnyOf}_{i_j} \land \text{AnyOf}_{i_m}$
   9. $Q \leftarrow Q \cup Z3\text{-request(anyConstraint)}$
10. For each $\text{AllOf}_k$ in $[\text{AllOf}_{i_{j_1}}, \text{AllOf}_{i_{j_k}}, ..., \text{AllOf}_{i_{j_l}}]$ in $rt_i$, do
   11. $allConstraint \leftarrow ruleConstraint \land \text{AnyOf}_{i_1} \land \neg \text{AllOf}_{i_{j_1}} \lor \text{AllOf}_{i_{j_k}}$
       $\neg \ldots \lor \text{AllOf}_{i_{j_l}} \land \text{AnyOf}_{i_m}$
   12. $Q \leftarrow Q \cup Z3\text{-request(anyConstraint)}$
13. $return Q$

If there are $m$ $\text{AnyOf}$ clauses, then there should be $m+1$ expression such that one makes all of them $true$ and the rest makes each of them evaluate to $false$ while the other evaluates to $true$. In addition, for each $\text{AnyOf}$ clause, if there are $n$ $\text{AllOf}$ clauses, then there should be $n+1$ expression such that one makes all of them evaluate to $false$ and the rest makes each of them evaluate to $true$ while others evaluate to $false$. 
4.16 FDC constraint and test generation for CRC

**Algorithm 13: generateTestsForCCA(P)**

*Input: Policy* $P = < PT, RCA, \{r_1, r_2, ..., r_n\}>$

*Output: A set of access requests* $Q$

1. $Q \leftarrow \emptyset$
2. $Constraint1 \leftarrow PT$
3. $Constraint2 \leftarrow PT$
4. $p \leftarrow$ find the position of permit Rule and deny Rule such that at least one of them evaluates to true and other evaluates to true or error
5. $d \leftarrow$ find first deny Rule after $p$
6. if such $p$ and $d$ does not exist
7. go to step 11
8. $Constraint1 \leftarrow Constraint1 \land r_t \land r_c$
9. $Constraint1 \leftarrow Constraint1 \land Error(rt \land r_c)$
10. $Q \leftarrow Q \cup Z3\text{-request}(Constraint1)$
11. $Constraint2 \leftarrow Constraint2 \land Error \text{ for all the rules } \{r_1 ... r_n\}$
12. $Q \leftarrow Q \cup Z3\text{-request}(Constraint2)$
13. Return $Q$

The sufficient condition to kill CCA mutants is to make target and condition of a pair of the rules evaluate to true such that one of them is permit rule and other is deny rule. In addition, all the rules that can be made evaluated to an error should be evaluated to an error.
CHAPTER 5

Quantitative Analysis

This section presents our experiment that aims to evaluate the cost-effectiveness of the afore-mentioned testing methods by comparing them to SMT. As mutation score is commonly used as the main indicator of effectiveness for a testing method, we measure cost-effectiveness by comparing mutation score with test suite size i.e how many mutants are killed given the test suite size. Further, the test suite size reflects the average time of test execution. We will also discuss test generation time to reflect the run-time efficiency of testing methods.

5.1 Experiment Setup

The mutation-based test generation and all coverage-based test generation methods discussed in this work are implemented in open source tool - XPA (XACML Policy Analyzer) [15] which is based on Balana [13] - an open source implementation of XACML 3.0. Our experiment was performed on a 64bit Ubuntu laptop with 8th Generation Intel® Core™ i7-8550U Processor (1.80GHz 8MB) and 16.0GB DDR4 2400MHz. The experiments use various XACML 3.0 policies with different levels of complexity as shown in Table 5.1. K-Market is the sample policy from the Balana [13]. Sample, Sample-fa, and Sample-dup policies are created by us for this research to cover the language feature of XACML policies not covered by other policies in the literature. All other policies are from the literature. iTrust-X and fedora-rule3-X are synthesized from iTrust and fedora policies.
respectively to study behavior in larger sized policies. The number of the rules ranges from 3 to 1,280 as depicted in Table 5.1 where ‘#’ represents ‘number of’ i.e. ‘#Rules’ means ‘number of the rules’ and LOC represents the line of code (markup) in the corresponding policies.

The experiment involves generating mutants of each policy by using the mutation operators in Table 3.1. Each mutation operator may generate several mutants for a given policy. For example, given a policy with \( n \) rules, CRE (Change Rule Effect) creates \( n \) mutants because it creates a mutant by changing the effect of each rule. The mutation operators in Table 3.1 is based on the operators in the mutant generator, XACMUT [4].

The next activity is to identify the number of equivalent mutants and non-equivalent mutants. As we applied strong mutation to generate SMT test suite, the number of mutants killed from the strong mutation-based test suite (SMT) are non-equivalent mutants while live mutants are equivalent mutants. Hence, we generate SMT test suite and run against the mutants to identify the equivalent and non-equivalent mutants. As a result, the analysis excludes mutants that are equivalent to their original policy. For example, the rule combining algorithms permit-overrides and deny-overrides make no difference with respect to a policy with permit-only (or deny-only) rules. Hence, a policy which only has permit rules and its RCA changed from permit-overrides to deny-overrides are equivalent. Our prior work on the formalization of semantic differences between combining algorithms [26] provides descriptions about the conditions under which two combining algorithms are equivalent. Table 5.1 lists the number of mutants for each subject policy.

As a next step, we generate test suites for each of the coverage-based test generation methods for each of the subject policies and run that test suite against each policy and
record the actual response of each test. We know each recorded response for each request (test case) is the correct response from the policy for this test case. Hence, it can be used as the Oracle value of this test case when the mutants are tested later. Finally, we run the test suite of each test generation method against mutants. Since mutants represent the faults that are likely to occur in XACML policies, mutation score is considered the indicator of the fault-detection capability, as commonly used by the software testing community [16].

Table 5.1: Policies used for the experiment

<table>
<thead>
<tr>
<th>No</th>
<th>Subject Policy</th>
<th>LOC</th>
<th>#Rules</th>
<th>#Equivalent Mutants</th>
<th>#Non-equivalent Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Conference [21]</td>
<td>228</td>
<td>15</td>
<td>1</td>
<td>91</td>
</tr>
<tr>
<td>2</td>
<td>fedora-rule3(^1)</td>
<td>226</td>
<td>12</td>
<td>1</td>
<td>87</td>
</tr>
<tr>
<td>3</td>
<td>K-Market-blue</td>
<td>84</td>
<td>4</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>[13]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>K-Market-silver</td>
<td>58</td>
<td>3</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>[13]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>K-Market-gold</td>
<td>106</td>
<td>5</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>[13]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Sample</td>
<td>152</td>
<td>6</td>
<td>1</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>Sample-fa</td>
<td>114</td>
<td>4</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>Sample-dup</td>
<td>80</td>
<td>4</td>
<td>1</td>
<td>33</td>
</tr>
</tbody>
</table>

\(^{1}\) [http://www.fedora.info](http://www.fedora.info)
The following section presents the results of the conducted experiment.

5.2 Results

5.2.1 Test Suite Size and Test Generation Time

Table 5.2 presents the number of tests generated for each subject policy for each testing method. Typically, the test suite for NO-SMT has more tests. The test suite size of MC/DC and SMT are nearly the same however SMT test suite is not larger than MC/DC test suite for all the subject policies. The size of each RC test suite is the number of the rule in the policy and it is always the smallest. The difference in test suite size between NE-DC and DC as well as NE-MC/DC and MC/DC represents the error tests. Hence, there are only a few error tests. For fedora with 12 rules has six error tests, while that of iTrust3-5 has only one error tests for the policy with 320 rules. The reason is that most of the rules in the iTrustX policy are defined over the same set of attributes - when a test makes one rule evaluate to indeterminate, it will do the same for all other similar rules. As mentioned before, SMT is computationally costly because of the involved optimization. It took SMT

---

2 http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start
36.02 hours (129655675 ms in Table 5.3) to complete the test generation for \textit{iTrust3-10}. The estimated \textit{SMT} test generation time for \textit{iTrust3-20} is nearly 10 days. As a result, we decided not to run SMT for iTrust-20. However, from \textit{NO-SMT}, we know that for each mutant, there is at least one test that kills it.
Table 5.2: Number of tests generated

<table>
<thead>
<tr>
<th>Subject Policy</th>
<th>RC</th>
<th>NE-DC</th>
<th>DC</th>
<th>NE-MC/DC</th>
<th>MC/DC</th>
<th>SMT</th>
<th>NO-SMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>coference3</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>78</td>
</tr>
<tr>
<td>fedora-rule3</td>
<td>12</td>
<td>19</td>
<td>25</td>
<td>24</td>
<td>30</td>
<td>23</td>
<td>62</td>
</tr>
<tr>
<td>kMarket-blue-policy</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>8</td>
<td>11</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>kMarket-gold-policy</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>kMarket-sliver-policy</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>9</td>
<td>13</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>sample</td>
<td>6</td>
<td>13</td>
<td>19</td>
<td>15</td>
<td>21</td>
<td>16</td>
<td>51</td>
</tr>
<tr>
<td>sample-fa</td>
<td>4</td>
<td>9</td>
<td>15</td>
<td>10</td>
<td>16</td>
<td>11</td>
<td>30</td>
</tr>
<tr>
<td>sample-dup</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>9</td>
<td>12</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>fedora-rule3-2</td>
<td>32</td>
<td>39</td>
<td>45</td>
<td>64</td>
<td>70</td>
<td>62</td>
<td>162</td>
</tr>
<tr>
<td>fedora-rule3-3</td>
<td>212</td>
<td>219</td>
<td>225</td>
<td>244</td>
<td>250</td>
<td>242</td>
<td>702</td>
</tr>
<tr>
<td>iTrust3</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>196</td>
<td>197</td>
<td>196</td>
<td>387</td>
</tr>
<tr>
<td>iTrust3-5</td>
<td>320</td>
<td>321</td>
<td>322</td>
<td>982</td>
<td>983</td>
<td>982</td>
<td>1923</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>iTrust3-10</td>
<td>640</td>
<td>641</td>
<td>642</td>
<td>1964</td>
<td>1965</td>
<td>1964</td>
<td>3843</td>
</tr>
<tr>
<td>iTrust3-20</td>
<td>1280</td>
<td>1281</td>
<td>1282</td>
<td>3928</td>
<td>3929</td>
<td>-</td>
<td>7683</td>
</tr>
</tbody>
</table>
Table 5.3 shows the test generation time for each of the method for each of the subject policies. The RC takes minimal test generation time as it has lesser number of tests. The test generation time is roughly proportional to test suite size except for the SMT because of the involved optimization of a test suite. All the testing methods are scalable except for the SMT. As test generation time for iTrust3-10 is nearly one and half days, we decided not to run SMT for iTrust3-20 because approximated time is nearly 10 days and this shows that SMT is not scalable because of the involved mutation analysis for optimization of a test suite size. However, for a policy with roughly hundred or lesser rules, SMT test generation time is comparable to that of other testing methods.
Table 5.3: Test generation time (in milliseconds)

<table>
<thead>
<tr>
<th>Subject Policy</th>
<th>RC</th>
<th>NE-DC</th>
<th>DC</th>
<th>NE-MC/DC</th>
<th>MC/DC</th>
<th>SMT</th>
<th>NO-SMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>conference3</td>
<td>488</td>
<td>438</td>
<td>408</td>
<td>689</td>
<td>716</td>
<td>2847</td>
<td>2019</td>
</tr>
<tr>
<td>fedora-rule3</td>
<td>328</td>
<td>548</td>
<td>686</td>
<td>652</td>
<td>536</td>
<td>2222</td>
<td>1628</td>
</tr>
<tr>
<td>kMarket-blue-policy</td>
<td>96</td>
<td>176</td>
<td>252</td>
<td>210</td>
<td>307</td>
<td>467</td>
<td>519</td>
</tr>
<tr>
<td>kMarket-gold-policy</td>
<td>71</td>
<td>154</td>
<td>221</td>
<td>150</td>
<td>221</td>
<td>343</td>
<td>385</td>
</tr>
<tr>
<td>kMarket-sliver-policy</td>
<td>130</td>
<td>252</td>
<td>328</td>
<td>231</td>
<td>335</td>
<td>704</td>
<td>626</td>
</tr>
<tr>
<td>sample</td>
<td>174</td>
<td>358</td>
<td>506</td>
<td>435</td>
<td>557</td>
<td>1835</td>
<td>1309</td>
</tr>
<tr>
<td>sample-fa</td>
<td>101</td>
<td>241</td>
<td>569</td>
<td>276</td>
<td>488</td>
<td>862</td>
<td>763</td>
</tr>
<tr>
<td>sample-dup</td>
<td>117</td>
<td>201</td>
<td>270</td>
<td>237</td>
<td>323</td>
<td>669</td>
<td>593</td>
</tr>
<tr>
<td>fedora-rule3-2</td>
<td>847</td>
<td>1062</td>
<td>1290</td>
<td>1790</td>
<td>1935</td>
<td>14853</td>
<td>4534</td>
</tr>
<tr>
<td>fedora-rule3-3</td>
<td>5368</td>
<td>7779</td>
<td>8120</td>
<td>9467</td>
<td>10304</td>
<td>930226</td>
<td>22625</td>
</tr>
<tr>
<td>iTrust3</td>
<td>1644</td>
<td>1555</td>
<td>3511</td>
<td>5923</td>
<td>8349</td>
<td>171716</td>
<td>10363</td>
</tr>
<tr>
<td>iTrust3-5</td>
<td>8711</td>
<td>8884</td>
<td>24415</td>
<td>49093</td>
<td>64749</td>
<td>15875846</td>
<td>48091</td>
</tr>
<tr>
<td>iTrust3-10</td>
<td>13890</td>
<td>19089</td>
<td>84304</td>
<td>146987</td>
<td>211230</td>
<td>129655675</td>
<td>176758</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td>iTrust3-20</td>
<td>45908</td>
<td>422653</td>
<td>562857</td>
<td>499790</td>
<td>696428</td>
<td>*</td>
<td>709927</td>
</tr>
</tbody>
</table>
The next section presents the mutation score and summarizes whether the mutants type (based on mutation operators) are killed or not by each testing method.

5.2.2 Fault Detection Capability

Table 5.4 presents the mutation score for each testing method for each subject policy. The mutation score for SMT and NO-SMT are 100% for all the policies, so we did not include them in Table 5.4. It should be noted that we did not generate and run SMT test suite for iTrust3-20. However, from NO-SMT, we know that for each mutant, there is at least one test that kills it. So, it will be 100% as well.
Table 5.4: Mutation Scores

<table>
<thead>
<tr>
<th>Subject Policy</th>
<th>RC</th>
<th>NE-DC</th>
<th>DC</th>
<th>NE-MC/DC</th>
<th>MC/DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>conference3</td>
<td>75.82</td>
<td>75.82</td>
<td>78.02</td>
<td>97.8</td>
<td>100</td>
</tr>
<tr>
<td>fedora-rule3</td>
<td>64.37</td>
<td>85.06</td>
<td>85.06</td>
<td>88.51</td>
<td>88.51</td>
</tr>
<tr>
<td>kMarket-blue-policy</td>
<td>81.48</td>
<td>96.3</td>
<td>96.3</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>kMarket-gold-policy</td>
<td>80.95</td>
<td>95.24</td>
<td>95.24</td>
<td>95.24</td>
<td>95.24</td>
</tr>
<tr>
<td>kMarket-silver-policy</td>
<td>81.25</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>sample</td>
<td>69.09</td>
<td>87.27</td>
<td>90.91</td>
<td>92.73</td>
<td>96.36</td>
</tr>
<tr>
<td>sample-fa</td>
<td>66.67</td>
<td>85.71</td>
<td>90.48</td>
<td>92.86</td>
<td>97.62</td>
</tr>
<tr>
<td>sample-dup</td>
<td>48.48</td>
<td>78.79</td>
<td>84.85</td>
<td>90.91</td>
<td>96.97</td>
</tr>
<tr>
<td>fedora-rule3-2</td>
<td>36.71</td>
<td>55.07</td>
<td>55.07</td>
<td>56.52</td>
<td>56.52</td>
</tr>
<tr>
<td>fedora-rule3-3</td>
<td>27.62</td>
<td>51.13</td>
<td>51.13</td>
<td>51.46</td>
<td>51.46</td>
</tr>
<tr>
<td>iTrust3</td>
<td>42.67</td>
<td>58.22</td>
<td>58.22</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>iTrust3-5</td>
<td>42.82</td>
<td>57.36</td>
<td>57.36</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>iTrust3-10</td>
<td>42.83</td>
<td>57.25</td>
<td>57.25</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>iTrust3-20</td>
<td>42.84</td>
<td>57.19</td>
<td>57.19</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
Table 5.5 presents the type of mutants that could not be killed by each testing method. In Table 5.5, the results for three kMarket policies are combined into one. Further, iTrust3 and fedora-rule3-2 can represent other policies in the group of iTrust-X and fedora-rule3-X from Table 5.1, hence only iTrust3 and fedora-rule3-2 are included. The absence of mutation operator in Table 5.5 represents that all the mutants of the corresponding type are killed by that method for that policy or the mutation operator is not applicable to the policy.

The mutation operator without asterisk implies all the mutants of the type are live. For example, RTT specifies, all the mutants of type RTT are not killed. The mutation operator with an asterisk means some are not killed while some are not. Single asterisk implies less than 50% of the type of mutant are not killed and double asterisk implies half or more percentage of mutants of the type are not killed.
Table 5.5: Live Mutants

<table>
<thead>
<tr>
<th>Subject</th>
<th>Policy</th>
<th>Testing Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RC</td>
<td>NE-DC</td>
</tr>
<tr>
<td>Sample</td>
<td>RTT **</td>
<td>RTT **</td>
</tr>
<tr>
<td></td>
<td>RCT</td>
<td>RTT *</td>
</tr>
<tr>
<td></td>
<td>PTF</td>
<td>RTT **</td>
</tr>
<tr>
<td></td>
<td>RPTE **</td>
<td>RPTE **</td>
</tr>
<tr>
<td></td>
<td>CRC ** {FA, DUP, PUD}</td>
<td></td>
</tr>
<tr>
<td>Sample-dup</td>
<td>CRE*</td>
<td>CRE*</td>
</tr>
<tr>
<td></td>
<td>RTT</td>
<td>CRE*</td>
</tr>
<tr>
<td></td>
<td>RTF*</td>
<td>RTF*</td>
</tr>
<tr>
<td></td>
<td>RCT</td>
<td>RTF*</td>
</tr>
<tr>
<td></td>
<td>RER*</td>
<td>RCT</td>
</tr>
<tr>
<td></td>
<td>RPTE**</td>
<td>RER*</td>
</tr>
<tr>
<td></td>
<td>CRC</td>
<td>RPTE**</td>
</tr>
<tr>
<td>Sample-fa</td>
<td>RTT**</td>
<td>RPTE**</td>
</tr>
<tr>
<td></td>
<td>RCT**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PTT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FDR</td>
<td>FDR</td>
</tr>
<tr>
<td></td>
<td>RPTE**</td>
<td>FDR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FDR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FDR</td>
</tr>
<tr>
<td></td>
<td>CRC** {DO, ODO, PUD}</td>
<td>CRC* {DO, ODO}</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Conference3</td>
<td>RPTE**</td>
<td>RPTE**</td>
</tr>
<tr>
<td></td>
<td>CRC* {FA, DUP}</td>
<td>CRC* {FA, DUP}</td>
</tr>
<tr>
<td>fedora-rule3-2</td>
<td>RTT**</td>
<td>RTF**</td>
</tr>
<tr>
<td></td>
<td>RTF**</td>
<td>RCT**</td>
</tr>
<tr>
<td></td>
<td>RER**</td>
<td>RPTE**</td>
</tr>
<tr>
<td></td>
<td>RPTE**</td>
<td>CRC {FA, PUD}</td>
</tr>
<tr>
<td>Kmarket</td>
<td>RTT**</td>
<td>RTT*</td>
</tr>
<tr>
<td></td>
<td>RCT*</td>
<td>RCT*</td>
</tr>
<tr>
<td></td>
<td>PTT</td>
<td>PTT</td>
</tr>
<tr>
<td></td>
<td>RPTE**</td>
<td>RPTE**</td>
</tr>
<tr>
<td></td>
<td>CRC** {FA, PUD}</td>
<td></td>
</tr>
<tr>
<td>iTTrust</td>
<td>RTT</td>
<td>CRC</td>
</tr>
<tr>
<td></td>
<td>CRC</td>
<td>RPTE**</td>
</tr>
</tbody>
</table>
The mutation score increases for methods from left to right in Table 5.4 and live mutants type decreases as we go from left to right in Table 5.5. The mutation scores ranged from 27% to 81% for the RC tests and Table 5.5 shows that it could not kill a majority of the mutants. The mutation scores for NE-DC, DC, MC/DC and NE-MC/DC ranged from 51% to 100%. However, this does not mean that they have similar fault detection capability. It is clear from Table 5.4 that all the policies have greater (if not the same) MC/DC (or NE-MC/DC) mutation score than that from DC (or NE-DC). It is also supported from the result in Table 5.5 which implies that the difference between fault detection capability among MC/DC and DC is the ability of MC/DC to detect the RPTE mutant type.

It is also noteworthy that error tests (from NE-DC and NE-MC/DC) do not always necessarily contribute to fault detection. However, they are crucial for detecting faults resulted from CRC mutation operator as well as RCT mutation operators. Hence, the error version of the testing method is recommended for higher fault detection capability. Further, if we do not consider fedora-rule3-2 policy, Table 5.4 shows that the mutation score of MC/DC is above 90% in most of the policies. Further, Table 5.2 shows that MC/DC and SMT have comparable tests size and Table 5.3 shows that test generation time of MC/DC is much less than SMT. These data not necessarily but may lead to infer that MC/DC is runtime efficient and cost-effective test suite for achieving above 90% mutation score and provide high-quality assurance of XACML policies. However, we know from the fault detection condition that MC/DC does not explicitly satisfy the propagation constraint for fault detection. As a result, if we increase the size of the policy where reachability and necessity constraint does not necessarily make the propagation constraint to be true like in fedora-rule3, the mutation score of MC/DC dropped from 88% to 51%.
Hence, though $MC/DC$ is likely to achieve good mutation score in many policies, it cannot assure that it will always lead to the high assurance of the XACML policies because it explicitly satisfies only the reachability and necessity constraint. However, it should also be mentioned that $MC/DC$ satisfies the reachability and necessity constraint of the majority of the faults that could be introduced by mutation operators in Table 3.1. Hence, we could qualify $MC/DC$ as near to weak mutation-based test generation method but that is not the case with $DC$ and $RC$. In fact, $RC$ is far from satisfying necessity constraint for many of the faults in Table 3.1.

5.2.3 Cost Effectiveness

While mutation score is a good indicator of the fault detection capability of a testing method, it does not account for cost-effectiveness of the test suite i.e the number of mutants killing capability of each test in a test suite. We consider the average number of Mutants Killed Per Test (MKPT) as the indicator for cost-effectiveness.
Table 5.6: MKPT Scores

<table>
<thead>
<tr>
<th>Subject</th>
<th>RC</th>
<th>NE-DC</th>
<th>DC</th>
<th>NE-MC/DC</th>
<th>MC/DC</th>
<th>SMT</th>
<th>NO-SMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>conference3</td>
<td>4.6</td>
<td>4.6</td>
<td>4.44</td>
<td>3.71</td>
<td>3.64</td>
<td>3.64</td>
<td>1.17</td>
</tr>
<tr>
<td>fedora-rule3</td>
<td>4.67</td>
<td>3.89</td>
<td>2.96</td>
<td>3.21</td>
<td>2.57</td>
<td>3.78</td>
<td>1.4</td>
</tr>
<tr>
<td>kMarket-blue-policy</td>
<td>5.5</td>
<td>3.71</td>
<td>2.6</td>
<td>3.38</td>
<td>2.45</td>
<td>3.86</td>
<td>1.29</td>
</tr>
<tr>
<td>kMarket-gold-policy</td>
<td>5.67</td>
<td>3.33</td>
<td>2.22</td>
<td>3.33</td>
<td>2.22</td>
<td>4.2</td>
<td>1.31</td>
</tr>
<tr>
<td>kMarket-sliver-policy</td>
<td>5.2</td>
<td>3.56</td>
<td>2.46</td>
<td>3.56</td>
<td>2.46</td>
<td>4</td>
<td>1.39</td>
</tr>
<tr>
<td>sample</td>
<td>6.33</td>
<td>3.69</td>
<td>2.63</td>
<td>3.4</td>
<td>2.52</td>
<td>3.44</td>
<td>1.08</td>
</tr>
<tr>
<td>sample-fa</td>
<td>7</td>
<td>4</td>
<td>2.53</td>
<td>3.9</td>
<td>2.56</td>
<td>3.82</td>
<td>1.4</td>
</tr>
<tr>
<td>sample-dup</td>
<td>4</td>
<td>3.71</td>
<td>2.8</td>
<td>3.33</td>
<td>2.67</td>
<td>4.12</td>
<td>1.43</td>
</tr>
<tr>
<td>fedora-rule3-2</td>
<td>2.38</td>
<td>2.92</td>
<td>2.53</td>
<td>1.83</td>
<td>1.67</td>
<td>3.34</td>
<td>1.28</td>
</tr>
<tr>
<td></td>
<td>1.21</td>
<td>2.16</td>
<td>2.11</td>
<td>1.95</td>
<td>1.91</td>
<td>3.83</td>
<td>1.32</td>
</tr>
<tr>
<td>----------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>fedora-rule3-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iTrust3</td>
<td>3</td>
<td>4.03</td>
<td>3.97</td>
<td>2.3</td>
<td>2.28</td>
<td>2.3</td>
<td>1.16</td>
</tr>
<tr>
<td>iTrust3-5</td>
<td>3</td>
<td>4.01</td>
<td>3.99</td>
<td>2.28</td>
<td>2.28</td>
<td>2.28</td>
<td>1.17</td>
</tr>
<tr>
<td>iTrust3-10</td>
<td>3</td>
<td>4.03</td>
<td>3.996</td>
<td>2.282</td>
<td>2.28</td>
<td>2.28</td>
<td>1.16</td>
</tr>
<tr>
<td>iTrust3-20</td>
<td>3</td>
<td>4.001</td>
<td>3.998</td>
<td>2.281</td>
<td>2.28</td>
<td>-</td>
<td>1.16</td>
</tr>
</tbody>
</table>
Table 5.6 shows MKPT scores for various testing methods discussed in this work. Further, it suggests that RC has good MKPT scores and in many policies, it even achieved the highest scores among other testing methods. However, it does not mean RC will have the best cost-effectiveness in all policies as it has lower MKPT scores in iTrust-X and fedora-rule3-X. NO-SMT, on the other hand, could achieve perfect mutation score but has lowest MKPT score in all the subject policies. DC (or NE-DC) has better MKPT than that of MC/DC (or NE-MC/DC). SMT and MC/DC have similar MKPT scores in many policies, however, SMT has either similar or better MKPT scores in all the subject policies than MC/DC. Hence, this shows that MC/DC is nearly as cost-effective as near-optimal test suite from SMT and so is the decision coverage. In fact, in some policies decision coverage (specifically NE-DC) has better MKPT score than SMT in many policies. However, they do not assure perfect mutation score in many policies and could not always provide high-quality assurance of the XACML policies.

5.3 Threats to Validity

There are many threats to validity. First, XPA is based on the Balana [13] – OASIS implementation of XACML and hence our results depend on the proper implementation of Balana. While analyzing some of the results when executing policy against tests, we have found some inconsistencies between XACML specification and results from the Balana. Hence, the error in Balana implementation could propagate into the experimental results that we presented in this chapter. Second, the subject policies do not necessarily represent all possible real-world XACML policies. Some of them originate from real XACML systems, others are demonstrating examples (e.g., kMarket) or synthesized (e.g., iTrustX, fedora-rule3-X). We have developed some sample policies (e.g., sample, sample-fa) to
capture the features not covered by policies in the literature of XACML. However, the
subject policies used in this research might still have not covered all language features of
XACML. Third, there is the possibility of different implementations for the coverage
criteria. Our result is based on the implementation of coverage-based test generation
methods in XPA. While we believe the proposed test generation algorithms reflect the
essential fault detection capabilities of the corresponding coverage criteria, different test
generation algorithms may lead to slightly different mutation scores and MKPT scores.
Fourth, all testing methods use Z3 as the constraint solver. Z3 is currently one of the most
efficient constraint solvers available. As different constraint solvers may result in different
attribute values for the same constraint, using another constraint solver in the
implementation may produce slightly different test suites and, thus, lead to slightly
different mutation scores. Fifth, although the 14 mutation operators have represented a
great variety of possible faults in XACML3.0 policies, they do not necessarily cover all
possible faults in real-world XACML3.0 policies. In addition, several mutation operators
in the literature [7] are not yet implemented in this work, including RUF (RemoveUnique-
nessFunction), AUF (AddUniquenessFunction), CNOF (Change-NOF-Function), and CLF
(ChangeLogicalFunction). Nevertheless, the formalization of the fault detection conditions
of incorrect rule targets and conditions has provided a foundation for dealing with these
types of mutants.
CHAPTER 6

Conclusions

6.1 Summary

We have presented the approach to strong mutation testing of XACML 3.0 policies and have formalized the strong fault detection conditions based on a comprehensive fault model of XACML 3.0 policies. On the one hand, these strong fault detection conditions have been exploited to generate near-optimal test suites of specific policies. On the other hand, the strong fault detection condition gave us insight into the fault detection capabilities of existing testing methods. It is clear from the fault detection condition that MC/DC is near to weak mutation-based test generation method because it satisfies the reachability and necessity constraint for most of the faults in the fault model. However, that is not the case with DC and RC. In fact, RC is far from satisfying necessity constraint for many of the faults. This suggests that though MC/DC is likely to achieve good mutation score in most of the policies, it still could not always assure high-quality of XACML policies as it does not explicitly satisfy the propagation constraint.

Further, the generation of the near-optimal test suite has enabled us to perform quantitative evaluations on the cost-effectiveness of the testing methods of XACML policies. The results from the quantitative section suggest that MC/DC is nearly as cost-effective as near-optimal test suite from SMT. SMT has perfect fault detection capability and good cost-effectiveness, however, it may be practically infeasible to apply for larger
policies with a larger number of the rules. The \textit{MC/DC} including \textit{DC} and \textit{RC} is scalable and cost-effective as close to the optimal test suite, however, they could not always assure the high fault detection. Although \textit{MC/DC} achieved good mutation score in most of the policies, it performed badly in some of the policies. As a result, if perfect fault detection capability is required and test suite size is not of prime concern, then we can use the \textit{NO-SMT} test suite. Hence, it is the trade-off between fault detection capability, cost-effectiveness and test generation time.

The formulated fault detection conditions for fault model considered in this work is not limited to the existing mutation operators. They provide theoretical guidelines for developing new testing methods and dealing with faults created by new mutation operators.

\textbf{6.2 Future work}

In this work, we have determined the overall fault detection capability of \textit{RC}, \textit{DC} and \textit{MC/DC} based on whether they satisfy the reachability and necessity constraints of all the mutation operators or not. However, this approach can be formally extended to qualitatively evaluate the fault detection capability of each of the testing method relating to fault type. The implication is that it neither needs to have a test suite nor the Oracle values.

One of the other future works is to look at opportunities for further optimizing the mutation-based test suite. Another research direction is to look for the opportunity to optimize NO-SMT test suite without involving costly mutation analysis for the optimization.

Finally, except the policies considered in this work, XPA would only be able to support the analysis of policies which have syntax like one in the policies used in this work.
However, XACML is a broad XML based specification which allows us to define various policies using various functions, syntax not involved in any of the policies in this work. As a result, extending the XPA to support various other functions and syntax in the policy specification to make it more robust XACML analyzer tool could be another work for the future.
REFERENCES


