Ionic: Collaborating Causal Fault Localization

by

Nadeegi Himahansi Fernando Dombawalage

B.Sc., Sri Lanka Institute of Information Technology, 2013

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty Applied Sciences

© Nadeegi Himahansi Fernando Dombawalage 2019
SIMON FRASER UNIVERSITY
Fall 2019

Copyright in this work rests with the author. Please ensure that any reproduction
or re-use is done in accordance with the relevant national copyright legislation.

Approval

Name:	**Nadeegi Himahansi Fernando Dombawalage**
Degree:	**Master of Science**
Title:	**Ionic: Collaborating Causal Fault Localization**
Examining Committee:	**Chair:** Arrvindh Shriraman Associate Professor
	Nick Sumner Senior Supervisor Associate Professor
	Keval Vora Supervisor Assistant Professor
	Tianzheng Wang Examiner Assistant Professor
Date Defended:	**September 23, 2019**

Abstract

Fault Localization Techniques (FLTs) assist in efficient localization of faults in software programs. However, conventional FLTs such as heuristic based Spectrum Based Fault Localization are mainly investigated on the individual statement granularity. At statement granularity, elements are considered in isolation. Hence, the information flow between them is ignored. This creates unfavourable circumstances when localizing defects with multiple collaborating faulty program elements. This thesis presents Ionic, an extension to an existing statement level FLT that incorporates information about how statements may collaborate to cause a bug. Ionic leverages the construction of causal models for collections of statements with respect to a bug. Ionic is then used within a new FLT, Hera, to efficiently localize bugs including multiple collaborating statements. Experimental results on real world software defects in the Defects4J benchmark suite show that Hera is better at localizing defects with multiple collaborating faulty elements.

Keywords: Fault Localization, Causal Models

Dedication

To my loving parents and to the memory of my beloved Namidu

Acknowledgements

First and foremost, I thank my senior supervisor, Dr. Nick Sumner. His support and the vast knowledge about the subject guided me in the right direction.

I extend my thanks to Dr. Keval Vora, Dr. Tianzheng Wang and Dr. Arrvindh Shriraman for serving as my examining committee. I highly appreciate Warunika Ranaweera's generous support in initiating the connections with SFU CS department.

I am indebted to my parents for their support, motivation, and sacrifices. I'm highly thankful to my sister and brother-in-law for giving me a home away from home in Canada. I am extremely grateful to late Thasan Namidu for his exceptional love, care, and sacrifices for me and my family. His demise left me devastated and created a difficult situation in the final stage of my Masters. I offer my heartfelt thanks to everyone who gave me the strength to handle it and reach this important milestone. Finally, I thank all the wonderful friends who made my graduate student life a pleasant experience.

Table of Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgements v

Table of Contents vi

List of Tables viii

List of Figures ix

1 Introduction 1

2 Motivation and Background 4

2.1 Motivating example 4

2.2 Spectrum Based Fault Localization (SBFL) 5

2.2.1 Motivating example on SBFL 6

2.3 Doric 6

2.3.1 Intuition 7

2.3.2 Causal Models 8

2.3.3 Motivating example on Doric 9

2.4 Limitations of existing FLTs 11

3 Approach 12

3.1 Collaborating Causality 12

3.2 Ionic 12

3.2.1 Intuition 13

3.2.2 Motivating example on IonicCL 14

3.3 Using Ionic for Fault Localization 15

3.3.1 All combinations 15

3.3.2 Hera - A novel FLT 15

4 Evaluation **20**
4.1 Research questions . . . 20
4.2 Benchmarks . . . 20
4.2.1 Definitions . . . 21
4.2.2 Limitations . . . 22
4.3 Metrics . . . 24
4.3.1 k@n . . . 24
4.3.2 Recall . . . 24
4.3.3 Precision . . . 25
4.4 Setup . . . 26
4.5 Evaluating Hera techniques . . . 26
4.5.1 Analysis . . . 26
4.6 Evaluating Hera vs other FLTs . . . 31
4.6.1 Obtaining other FLTs' results . . . 31
4.6.2 Converting Hera's ranked list of clusters to a ranked list of units . . 31
4.6.3 Results . . . 32
4.6.4 Analysis . . . 33
4.7 Case Studies . . . 38
4.7.1 Merge$_{Ionic}$-Rank$_{Ionic}$ vs Merge$_{Naive}$-Rank$_{Ionic}$ . . . 38
4.7.2 Merge$_{Ionic}$-Rank$_{Ionic}$ vs Merge$_{Approx}$-Rank$_{Ionic}$ vs Doric . . . 40
4.8 Threats to validity . . . 41

5 Related Work **43**

6 Conclusion and Future Work **45**
6.1 Conclusion . . . 45
6.2 Future work . . . 45

Bibliography **46**

Appendix A Proofs of Ionic **48**
A.1 Preliminary definitions . . . 48
A.2 Causal Models . . . 49
A.3 Syntax and semantics . . . 51
A.4 Probability theory . . . 53
A.5 IonicCL . . . 54
A.6 IonicCL proofs . . . 55

List of Tables

Table 4.1 Defects4J (version 1.1.0) and number of versions and LoC information. 'LoC' is average lines of code for each version version of the project [21] . 21

Table 4.3 $k@n_{\text{cluster-n}}$: k@n results as a percentage, when k $\in$ {1,2,3} and n $\in$ {1,3,5,10}. Each numeric cell interpreted as the % of versions localized k faulty units within top n clusters. #versions changes with the k, as we run the metric on versions with at least k faulty units (3.3). . . . 30

Table 4.2 Intra-ClusterPrecision$_{\text{cluster-n}}$: Intra-cluster precision for Hera techniques as a percentage, for n $\in$ {1,3,5,10}. Each numeric cell interpreted as, the geo metric mean of intra-cluster precision for top n clusters 30

Table 4.4 Geometric mean of cluster sizes for each technique. Values are rounded off to integer as cluster sizes would be in reality. 31

Table 4.5 $k@n_{\text{unit-n}}$: k@n results as a percentage, when k $\in$ {1, 2, 3, median, last} and n $\in$ {1,3,5,10}. Each numeric cell is interpreted as the % of versions localized k faulty units within top n clusters. #versions change with the k. 38

List of Figures

Figure 1.1 Foo.java 2
Figure 1.2 The ranked list created by an FLT for Foo.java in 1.1 2

Figure 2.1 Bar.java and its coverage matrix 5
Figure 2.2 Suspiciousness formulas for Ochiai and Dstar 6
Figure 2.3 Ranked list for Bar.java in Figure 2.1 using $Dstar^2$ suspiciousness scores. Standard ranking is used to rank, where ties share what would have been their first ordinal number to rank. 6
Figure 2.4 Coverage matrix 8
Figure 2.6 Ranked list for Bar.java in Figure 2.1 using DoricCL scores. Standard ranking is used to rank, where ties share what would have been their first ordinal number to rank. 9
Figure 2.5 Example of causal models for coverage matrix for Bar.java 2.1b . . 10

Figure 3.1 Comparison between DoricCL and IonicCL formulas 13
Figure 3.2 Ranked list for units and unit pairs for Bar.java in Figure 2.1 using IonicCL scores. Standard ranking is used to rank, where ties share what would have been their first ordinal number to rank. 14
Figure 3.3 Ranked list for unit clusters Bar.java in Figure 2.1 using Hera. Standard ranking is used to rank the clusters 19

Figure 4.1 Patch extract and ground truth table for Math-2 from Defects4J . . 21
Figure 4.2 Patch extract and ground truth table for Closure-15 from Defects4J 22
Figure 4.3 Number of versions against number of faulty units. Graph is read as, x faulty units are present in y versions 23
Figure 4.4 $\text{Recall}_{\text{cluster-n}}$: Cluster recall as a percentage against cluster rank for n $\in$ range [1..10]. Each graph shows one Hera technique. Each point is read as "At cluster x, y% of faulty units localized". The orange trend line shows the behaviour of the median of the versions. Likewise, blue and green show the behaviour of bottom and top 25% respectively. 28

Figure 4.5 $\text{Precision}_{\text{cluster-n}}$: Cluster precision as a percentage against cluster rank for n $\in$ range [1..10]. Each graph shows one Hera technique. Each point is read as "At cluster x, y% of clusters carries faulty unit s in them". The orange trend line shows the behaviour of the median of the versions. Likewise, blue and green show the behaviour of bottom and top 25% respectively. 29
Figure 4.6 Ranked list of clusters and faulty unit 32
Figure 4.7 Ranked list of clusters converted to units according to the three cases 32
Figure 4.8 $\text{Recall}_{\text{unit-n}}$: Unit recall as a percentage against unit rank for n $\in$ range [1..10]. Each graph shows one Hera technique. Each point is read as "At cluster x, y% of faulty units localized". The orange trend line shows the behaviour of the median of the versions. Likewise, blue and green show the behaviour of bottom and top 25% respectively. Blue and orange lines are overlapping in these cases 35
Figure 4.9 $\text{Precision}_{\text{unit-n}}$: Unit precision as a percentage against unit rank for n $\in$ range [1..10]. Each graph shows one Hera technique. Each point is read as "At cluster x, y% of faulty units localized". The orange trend line shows the behaviour of the median of the versions. Likewise, blue and green show the behaviour of bottom and top 25% respectively. Blue and orange lines are overlapping in these cases 36
Figure 4.10 Source code drawn from Chart-25 from Defects4J 39
Figure 4.11 Ranked list of clusters for Chart-25 using technique Merge_{Ionic}-Rank_{Ionic} 39
Figure 4.12 Ranked list of clusters for Chart-25 using technique Merge_{Naive}-Rank_{Ionic} 39
Figure 4.13 Simplified coverage matrix for Chart-25 39
Figure 4.14 Ranked list of pairs for Chart-25 40
Figure 4.15 Source code drawn from Lang-17 from Defects4J. 40
Figure 4.16 Ranked list of clusters for Lang-17 using technique Merge_{Ionic}-Rank_{Ionic} 41
Figure 4.17 Ranked list of clusters for Lang-17 using technique Merge_{Approx}-Rank_{Ionic} 41
Figure 4.18 Ranked list of clusters for Lang-17 using technique Doric 41

Figure A.1 Bar.java and its coverage matrix 49
Figure A.2 Coverage matrix 49
Figure A.3 Example of causal models for coverage matrix for Bar.java A.2 . . . 51

Chapter 1

Introduction

A software defect is a difference between the actual and the expected outcome of a software program. Software debugging is the process of locating and removing defects from software programs [5]. To debug a program, a developer runs test cases on the source code. Test cases might fail revealing that there may be defects in the source code. Then the developer examines the traces of failing test cases to locate the defects. Finally, the defects are fixed by the developer. Prior studies show that locating defects might take more effort than removing them [13].

Software fault localization is the process of identifying the locations of bugs that caused errors [12]. Automated fault localization is the process of using Fault Localization Techniques(FLTs) to locate faults efficiently with minimal human intervention. Suppose a program 'unit' is a source code element, such as a line of code. FLTs measure the 'suspiciousness' of each program unit to cause a failure by analysing their passing and failing test results. If a unit is executed more by failing test cases than passing test cases, then the unit is considered to be more suspicious. After calculating the suspiciousness for each element, an FLT outputs a ranked list of units. The first ranked unit in the list is the most suspicious and the last ranked unit is the least suspicious as a cause of the failure. The developer examines the ranked units in order and fixes units if needed.

Consider Foo.java shown in Figure 1.1. Suppose it has a defect in line 6. Suppose, the developer inputs the source code lines as units and their test results to an FLT. The FLT creates a ranked list of units as shown in Figure 1.2. This FLT suggests that line 6 is the most suspicious by ranking it as 1. Thus, the FLT ranked the faulty unit highly and could aid in debugging.

```
0     public int Foo (p, q){
1     if ( p > q )
2         p = p-100;
3     if ( q > 0 && q < 100 )
4         return p/q ;
5     if ( q < 200 )
6         return = q/p; //BUG
7     ..
8     }
```

Figure 1.1: Foo.java

Unit	Rank
6	1
5	2
2	3
3	4
1	5
4	6

Figure 1.2: The ranked list created by an FLT for Foo.java in 1.1

Suppose the source code had another fault at line 2. Suppose the faulty lines 2 and 6 together caused the failures. In such instances where two faulty units cause a failure together, we would prefer if an FLT can rank them together. This indicates their collaboration. But, existing FLTs cannot do that. This is because they focus on the unit granularity rather than on groups of units that may collaborate in causing a failure. Most prior work on FLTs have focused on investigating faults at the statement level [3]. This is mainly because the initial ideas of FLTs started at statement levels [8].

This limitation motivated us to find a new formula, IonicCL. IonicCL can calculate the likelihood of a set of units collaborating to cause a failure. Using this, we construct a new FLT, Hera following a simple process. We calculate the likelihood of unit pairs collaborating to cause the failure. We use those pairwise likelihoods to greedily partition the units executed in the program into likely collaborating clusters. Finally, we rank the partitions, based on the likelihood of the collaboration to cause the failure. We experimented with three levels of incorporating collaborative information in partitioning and ranking. Because Hera uses collaboration information while partitioning and produces a ranked list of collaborating partitions of units, it has the potential to provide better localization results when localizing defects with multiple collaborating lines when compared with traditional FLTs.

This thesis introduces IonicCL and Hera. IonicCL provides a formula for checking and quantifying the collaboration of multiple units to cause a failure. Hera, a novel FLT using IonicCL, outperforms conventional FLTs in localizing the defects with multiple faulty statements.

Chapter 2

Motivation and Background

In this section, first we present a motivating example. Then we describe some existing FLTs and how they produce their results for the example. Then, we discuss the limitations in the existing techniques.

2.1 Motivating example

Consider Bar.java at Figure 2.1. It has two faulty lines (line 4 and 8). Suppose the failure only occurs only when the faulty lines 4 and 8 are executed together in the same test case. Therefore, we can say lines 4 and 8 collaborate to cause the failure. The goal of an FLT is to rank lines 4 and 8 highly in their ranked list. Further, it is preferred if the FLT can rank lines 4 and 8 together to show their collaboration.

Suppose that we are only interested in suspiciousness of lines 2, 4, 6, 8 in this example (this avoids an overly complex example) and each line is a unit. Then, the set of units is U, where U = $\{u_2, u_4, u_6, u_8\}$. Suppose Bar.java's test suite is T, where T = $\{t_1, t_2, t_3, t_4, t_5, t_6\}$ and each $t_k \in$ T is a test case. FLTs need passing and failing test information as inputs. Suppose we ran the test cases and collected the test results as a *Coverage Matrix*.

Coverage Matrix (c) [12]: Suppose a the test suite (T) and a set of units (U). A coverage matrix is a Boolean matrix c of height $|T|$ and width $|U| + 1$, where for each $u_i \in U$ and $t_k \in T$,

$$c_{i,k} = \begin{cases} 1, & \text{if } t_k \text{ covers } u_i, \\ 0, & \text{otherwise} \end{cases}$$

Intuitively, each $c_{i,k}$ in the matrix marks whether u_i was executed in t_k or not. If u_i was executed, $c_{i,k}$ is marked as 1, otherwise it is marked as 0. An additional column e at last indicates the 'Error'. If the test case t_k failed, e_k is marked as 1,otherwise marked as 0. Suppose Figure 2.1b shows the coverage matrix for Bar.java's test results.

```
0    public int Bar(p, q){
1    if ( q > 500 && q < 1000 )
2        q = q-500;
3    if ( p > 500 && p < 1000 )
4        p = p-600;   //BUG
5    if ( q > 0 && q < 100 )
6        r = p/q ;
7    if ( q < 1000 )
8        s = q/p;     //BUG
9    ...
10   }
```

(a) Bar.java

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

(b) Coverage matrix

Figure 2.1: Bar.java and its coverage matrix

In the next two sections we will discuss how two types of existing FLTs would produce their results for this example.

2.2 Spectrum Based Fault Localization (SBFL)

SBFL is a family of heuristic based FLTs. It is found to be quite effective in fault localization [16, 17, 18]. An SBFL technique calculates the suspiciousness of a unit using a formula. The formula uses unit execution frequencies in test cases. If a unit is more often executed by failing test cases and less often by passing test cases, that unit is considered suspicious in terms of causing the failure. To create the ranked list, an SBFL uses the coverage matrix and the formula. There are many SBFL techniques available [1, 9, 20, 14]. Each technique in SBFL family has different suspiciousness formula. Figure 2.2 shows the formulas to calculate the suspiciousness of a unit s for two such FLTs, Ochiai [2], and $Dstar^*$ [20]. The definitions of their subformulas are as follows.

failed(s) - The number of times unit s was executed in failing test cases.

passed(s) - The number of times unit s was executed in passing test cases.

totalfailed - The number of failing test cases.

In these formulas the numerator is entirely dominated by the *failed(s)*. Therefore, the more the unit s was executed in failing test cases, the higher the suspiciousness. The Denominator is dominated by *passed(s)* and *totalfailed*. Therefore, the more the unit s was executed in passing test cases and not executed in failing test cases, the lower the suspiciousness.

$$Ochiai : Suspiciousness(s) = \frac{failed(s)}{\sqrt{totalfailed.(failed(s) + passed(s))}} \tag{2.1}$$

$$Dstar^* : Suspiciousness(s) = \frac{failed(s)^*}{passed(s) + totalfailed - failed(s)}, \text{ where * is a variable} > 0 \tag{2.2}$$

Figure 2.2: Suspiciousness formulas for Ochiai and Dstar

2.2.1 Motivating example on SBFL

Suspiciousness for each unit for the motivating example in Figure 2.1 can be calculated using $Dstar^*$ as below. We use $* = 2$ for this example as it is the most thoroughly explored value [16].
$S(2) = S(4) = \frac{failed(s)^2}{passed(s)+totalfailed-failed(s)} = \frac{2^2}{5+2-2} = 4/5 = 0.8$
$S(6) = 1/3 = 0.33$
$S(8) = 4/3 = 1.33$

Figure 2.3 shows the the ranked list created by $Dstar^2$ based on suspiciousness scores above.

Unit	Rank
8	1
2	2
4	2
6	4

Figure 2.3: Ranked list for Bar.java in Figure 2.1 using $Dstar^2$ suspiciousness scores. Standard ranking is used to rank, where ties share what would have been their first ordinal number to rank.

Units 8 is ranked 1. Therefore, 8 is the most suspicious as a cause of the failure. Units 2 and 4 are equally suspicious because they share the same rank. Therefore, in best case a developer is able to localize the bug to units 8 and 4 by examining the first two units. But in worst case the developer needs to examine top three to get to both faulty lines. The strength of localization got diluted because the non-faulty line 2 also carries the same suspiciousness as faulty line 4.

2.3 Doric

Doric [12] is a foundation for fault localization based on classical probability theory. It calculates the likelihood of a unit u_i causing the failure, when unit u_i was executed (*Causal*

Likelihood) using the formula 2.3. It then ranks the units based on the Causal Likelihood. When c is a coverage matrix and ρ_k is the number of units executed in test case k, Doric's Causal Likelihood formula is defined as,

$$DoricCL = P(H_i|u_i) = \frac{\sum_k [c_{i,k} e_k / (2^{\rho_k} - 1)]}{\sum_k c_{i,k}} \tag{2.3}$$

2.3.1 Intuition

A rearranged version of the formula is shown below.

$$P(H_i|u_i) = \sum_k \frac{c_{i,k} e_k}{2^{\rho_k} - 1} * \frac{1}{\sum_k c_{i,k}}$$

The intuition of DoricCL is explained as below. The intention of the of formula is to calculate the likelihood of u_i causing the failure based on passing and failing test case evidence in coverage matrix c.

Probability of u_i causing t_k to fail

The fraction shown in blue calculates the probability of u_i causing the the failure in test case t_k. The explanation as follows.

When ρ_k is the number of units executed in t_k, 2^{ρ_k} is the number of unit subsets that could have caused the *test result* of t_k. We remove the empty subset because, we assume at least one unit should be blamed to get a test result. Therefore, number of valid blamable subsets $= 2^{\rho_k} - 1$. Consider the t_2 in the motivating example. The number of valid subsets for t_2 is $2^3 - 1 = 7$. Valid subsets are $\{\{u_2\}, \{u_4\}, \{u_8\}, \{u_2, u_4\}, \{u_2, u_8\}, \{u_4, u_8\}, \{u_2, u_4, u_8\}$. This will be explained further using an illustration in section 2.3.2.

The coverage matrix c is a Boolean matrix. In c, $c_{i,k}$ is 1 if t_k executed u_i, otherwise $c_{i,k}$ is 0. A test case result, e_k is 1 if t_k failed, otherwise e_k is 0. According to this, $c_{i,k} e_k$ is 1 only if u_i was executed in t_k and t_k failed. If either u_i wasn't executed or t_k passed, $c_{i,k} e_k$ is 0.

Therefore, the fraction in blue evaluates to $1/(2^{\rho_k} - 1)$ if u_i executed in a failing t_k, otherwise the fraction is 0.

This $1/(2^{\rho_k} - 1)$ intuitively captures the probability of u_i causing the *failure* in t_k. The reasoning as follows. In probability theory, if there are n mutually exclusive possibilities, and there is no sufficient reason to believe one over the other, then their relative likelihoods are equal. The probability of one event happening becomes $1/n$ [6]. Similarly, there are $2^{\rho_k} - 1$ subsets of units that could be blamed for the failure of t_k. And we don't have enough evidence to believe one blamable subset over the other. Thus, the probability of any subset causing the failure is $1/(2^{\rho_k} - 1)$. Hence, probability of u_i causing the failure of t_k is $1/(2^{\rho_k} - 1)$.

Measuring the likelihood of u_i causing the failures in all test cases

The summation in brown combines the likelihoods of u_i causing all the test failures. So far, the formula only includes failing evidence. If u_i was executed in passing test cases, the overall likelihood should be further penalized. The summation in green counts the number of test cases in which u_i was executed. This includes the passing test cases if any. Therefore, dividing by this summation penalizes u_i's faultiness.

2.3.2 Causal Models

In this section we provide an illustration to explain how probability of u_i causing failure in t_k is $1/(2^{\rho_k} - 1)$. Consider the coverage matrix for the motivating example Bar.java.

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Figure 2.4: Coverage matrix

Causal models a capture the entire universe of possibilities that could have caused the failures. Suppose $j \in \mathbb{N}$ and m_j is a matrix. Then $m^j_{i,k}$ is the value of the cell located at the ith column and kth row of matrix m^j, where this value is in $\{1, 0, \bullet\}$. Rows represent test cases and the columns represent units. $m^j_{i,k} = 0$ denotes u_i was not executed and hence cannot be a cause of the error e. 1 denotes u_i was executed but was not a cause of e. $\bullet$ denotes u_i was executed by t_k and was a cause of e. Given a the test suite (T), set of units (U), test case result (e) and Coverage matrix (c), causal model is defined as below.

When T is a test suite and U a set of units, the set of models for a coverage matrix c is a set of matrices $M = m^1, ..., m^{|M|}$ of height $|T|$ and width $|U|$ satisfying:

$$m^j_{i,k} \in \begin{cases} \{1, \bullet\} & \text{if } c_{i,k}e_k = 1 \text{ and } i \neq |U|, \\ c_{i,k} & \text{otherwise} \end{cases}$$

where for each $t_k \in T$ there is some $u_i \in U$ such that $m^j_{i,k} = \bullet$. Intuitively, each causal model represents one possible way of causing the failures of test cases.

The number of possible causal models is $\prod_{k \in \tilde{T}} 2^{\rho_k} - 1$, where $\tilde{T} \subseteq T$ and $\forall t_k \in \tilde{T}, e_k = 1$. Thus, the number of possible causal models for motivating example is $(2^4 - 1) * (2^3 - 1) = 105$. Some of those causal models are shown in Figure 2.5.

Since the causal models capture the universe of causes of failures, a probability calculation can be framed as model counting problem as below.
The probability of u_i causing failure in $t_k = \frac{\#\text{Models } u_i \text{caused failure in } t_k}{\#\text{Total models}}$
The in the running example this behaves as follows.
The probability of u_2 causing failure in $t_1 = 7/105$
The probability of u_2 and u_4 together causing failure in $t_1 = 7/105$
The probability of u_2, u_4, u_6 and u_8 together causing failure in $t_1 = 7/105$
The probability of u_2 causing failure in t_3 together $= 15/105$
The probability of u_2, u_4 and u_8 causing failure in $t_3 = 15/105$

2.3.3 Motivating example on Doric

The Causal Likelihood for units for example Bar.java can be calculated DoricCL as below.
$P(H_2|u_2) = \frac{\sum_k \{c_{1,k} e_k/(2^{\rho_k}-1)\}}{\sum_k c_{1,k}} = \frac{\{1/(2^4-1)\}+\{1/(2^3-1)\}}{5} = 0.04$
$P(H_4|u_4) = P(H_2|u_2)\ P(H_6|u_6) = \frac{\{1/(2^4-1)\}}{2} = 0.02$
$P(H_8|u_8) = \frac{\{1/(2^4-1)\}+\{1/(2^3-1)\}}{3} = 0.07$

Figure 2.6 shows the the ranked list created by DoricCL based on Causal Likelihood scores above.

Unit	Rank
8	1
2	2
4	2
6	4

Figure 2.6: Ranked list for Bar.java in Figure 2.1 using DoricCL scores. Standard ranking is used to rank, where ties share what would have been their first ordinal number to rank.

Similar to $DStar^2$, Doric also only could rank faulty line 8 highly. The rank of faulty line 4 is shared with another non-faulty unit.

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
\bullet & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
\bullet & \bullet & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
\bullet & 1 & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
\bullet & \bullet & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
1 & \bullet & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
1 & \bullet & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
\bullet & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
\bullet & \bullet & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
\bullet & 1 & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
\bullet & \bullet & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
1 & \bullet & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
1 & \bullet & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
1 & 1 & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad \ldots\ldots
$$

Figure 2.5: Example of causal models for coverage matrix for Bar.java 2.1b

2.4 Limitations of existing FLTs

The motivating example with Bar.java has two collaborating faulty units, 2 and 8. But neither Doric or $Dstar^2$ could rank both units highly. Further, though it was preferable to rank 2 and 8 together to show their collaboration, none of these techniques perform rankings on unit *combinations*. The reason is they are unit level FLTs. Their 'Suspiciousness' or 'Causal Likelihood' formulas only calculate the causality of units in isolation. This is a limitation in existing FLTs. This motivates us to find a formula that can calculate collaborating causality of a *set of units* together. Then, using that formula, we can create a ranked list of *unit sets* that could have collaborated to cause a failure. The new formula is created to consider collaboration between units and their impact on failures. Thus, we expect it to rank collaborating units highly.

Chapter 3

Approach

In this chapter, first we define *Collaborating Causality* formally. Then we introduce the new formula IonicCL, which calculates the Causal Likelihood of a set of units. Then we introduce Hera, an FLT that uses IonicCL to create a ranked list of sets of units that could have collaborated to cause the error.

3.1 Collaborating Causality

The motivating example has two faulty units 4 and 8 that should be executed to cause the failure. Therefore, those multiple units are said to be collaborating to cause the failure. Formally, suppose a failing program has a set of units U. If there is a $\tilde{U} \subseteq$ U where all $u_i \in \tilde{U}$,

1. Must be executed to cause the failure
2. Must be fixed to fix the bug

, $\tilde{U}$ is a set of *Collaborating Causal Units* of the program.

For the motivating example Bar.java, we suppose units 4 and 8 are such Collaborating Causal Units.

3.2 Ionic

Ionic is an extended framework of Doric. The complete framework extension is available in Appendix A. The final outcome of Ionic framework is the IonicCL formula. In this section we present the intuition for extension of DoricCL to IonicCL. IonicCL measures the Collaborating Causality of a set of units. Formally, it calculates the Causal Likelihood of $\tilde{U}$ $\subseteq$ U causing the failure, when $\tilde{U}$ executed,

When $\tilde{U} \subseteq$ U,

$$IonicCL = P(H_{\tilde{U}}|u_{\tilde{U}}) = \frac{\sum_k \{\prod_{i \in \tilde{U}} c_{i,k} e_k / (2^{\rho_k} - 1)\}}{\sum_k \prod_{i \in \tilde{U}} c_{i,k} e_k} \tag{3.1}$$

3.2.1 Intuition

A rearranged version of the IonicCL formula is compared against DoricCL as shown in Figure 3.1. The intuition of IonicCL is explained while comparing it to DoricCL.

$$P(H_i|u_i) = \sum_k \frac{c_{i,k}e_k}{2^{\rho_k} - 1} * \frac{1}{\sum_k c_{i,k}} \qquad P(H_{\tilde{U}}|u_{\tilde{U}}) = \sum_k \frac{\prod_{i \in \tilde{U}} c_{i,k}e_k}{2^{\rho_k} - 1} * \frac{1}{\sum_k \prod_{i \in \tilde{U}} c_{i,k}e_k}$$

Figure 3.1: Comparison between DoricCL and IonicCL formulas

The intention of the of DoricCL is to calculate the likelihood of u_i causing the failure based on passing and failing test case evidence in coverage matrix. The intention of the of the IonicCL is to calculate the likelihood of $\tilde{U}$ causing the failure based on passing and failing test case evidence in coverage matrix.

Probability of units in $\tilde{U}$ causing t_k to fail

The fraction shown in blue in IonicCL calculates the probability of $\tilde{U}$ causing the failure in test case t_k.

The denominator for this fraction is same as DoricCL. This is because DoricCL's $2^{\rho_k} - 1$ captures all possible valid subsets of a set of units. This includes all the subsets with multiple units. Therefore, there is nothing to be extended in this portion. This is shown in the causal models in section 2.3.2.

The coverage matrix is a Boolean matrix. In c, $c_{i,k}$ is 1 if t_k executed u_i, otherwise $c_{i,k}$ is 0. A test case result, e_k is 1 if t_k failed, otherwise e_k is 0. According to this, $\prod_{i \in \tilde{U}} c_{i,k}e_k$ is 1 only if all $u_i \in \tilde{U}$ were executed in t_k and t_k failed. If any $u_i \in \tilde{U}$ wasn't executed or t_k passed, $\prod_{i \in \tilde{U}} c_{i,k}e_k$ is 0.

Therefore, the fraction in blue evaluates to $1/(2^{\rho_k} - 1)$, when all $u_i \in \tilde{U}$ were executed in a failing t_k, the fraction is 0. This behaviour is similar to Doric.

The fraction $1/(2^{\rho_k} - 1)$ intuitively captures the probability of $\tilde{U}$ causing the failure in t_k. The explanation is same as Doric because the valid number of subsets didn't change.

Measuring the likelihood of $\tilde{U}$ causing the failures in all test cases

The summation in brown combines the likelihoods of $\tilde{U}$ causing all the test failures. So far, the formula only includes failing evidence. If $u_i \in \tilde{U}$ were executed together in passing test cases, the overall likelihood should be further penalized. The summation in green counts the number of test cases where all $u_i \in \tilde{U}$ were executed together. This includes the passing test cases if any. Therefore, dividing by this summation penalizes $\tilde{U}$'s faultiness.

The detailed proofs for IonicCL can be found in Appendix A.

3.2.2 Motivating example on IonicCL

To show the Ionic's potential compared to existing unit level FLTs, we calculate the IonicCL for all the units and unit pairs for motivating example. Note that when the $|\tilde{U}| = 1$, IonicCL gives the same Causal Likelihood as DoricCL.

$P(H_{\{4,8\}}|u_{\{4,8\}}) = \frac{\{1/(2^4-1)\}+\{1/(2^3-1)\}}{2} = 0.1$

$P(H_{\{2,8\}}|u_{\{2,8\}}) = \frac{\{1/(2^4-1)\}+\{1/(2^3-1)\}}{3} = 0.07$

$P(H_8|u_8) = P(H_{\{2,8\}}|u_{\{2,8\}}) = 0.07$

$P(H_{\{2,4\}}|u_{\{2,4\}}) = \frac{\{1/(2^4-1)\}+\{1/(2^3-1)\}}{4} = 0.05$

$P(H_4|u_4) = \frac{\{1/(2^4-1)\}+\{1/(2^3-1)\}}{5} = 0.03$

$P(H_2|u_2) = P(H_4|u_4)$

$P(H_{6,8}|u_{6,8}) = P(H_{4,6}|u_{4,6}) = 0.03$

$P(H_6|u_6) = 0.02$

All the other pairs or units has a Causal Likelihood of 0 as they were not executed in any failing test case. Those pairs or units are not listed here.

Figure 3.2 shows the the ranked list created by IonicCL based on Causal Likelihood scores above.

Unit/Unit Pair	Rank
4,8	1
2,8	2
8	2
2,4	4
4	5
2	5
6,8	7
4,6	7
6	9

Figure 3.2: Ranked list for units and unit pairs for Bar.java in Figure 2.1 using IonicCL scores. Standard ranking is used to rank, where ties share what would have been their first ordinal number to rank.

IonicCL lists faulty pair {4,8} at rank 1, while any other pair or unit is ranked lower. Note that the faulty units 4 and 8 in isolation ranked lower than the pair {4,8}. This shows that IonicCL leveraged the collaboration information, leading it to suggest 2 *and* 8 collaborating has a higher chance of causing the failure rather than 2 *or* 8 in isolation. This was not possible with existing unit level FLTs.

3.3 Using Ionic for Fault Localization

IonicCL can calculate the Collaborating Causality of a set of units. Our goal is to create a ranked list of collaborating causal units sorted from most likely to least likely. There are numerous ways to use IonicCL to achieve this.

3.3.1 All combinations

The IonicCL equation (3.1) is capable of calculating collaborating causal likelihood of a set of units. Consider a set of units U. We could calculate IonicCL for the $\mathcal{P}(U)$ and rank them in descending order to get the desired ranked list. However this is intractable, as $|\mathcal{P}(U)| = 2^{|U|}$. Suppose units are lines of code in a program. For real world programs, |U| could be as large as 105k as shown in Figure 4.1. This makes this approach intractable. Hence, we propose the following greedy yet tractable approach, Hera. It calculates IonicCL for all unit pairs (*pairwise CLs*). The pairwise CLs are then used to consider and construct larger clusters that partition the units of a program. In next section, we describe Hera and discuss how it is tractable compared to the all combinations approach.

3.3.2 Hera - A novel FLT

Hera is presented in Algorithm 1. It shows the greedy partitioning of units, followed by scoring and ranking the partitions, leading to our goal. The parameter *Spectra* provides the coverage matrix to Hera. *shouldMerge* is a function used in partitioning. Partitioning is done using clustering. At the beginning, each unit is a cluster on its own. Iteratively, clusters are merged if merging them leads to a cluster with higher Collaborating Causality than individual clusters. Checking whether merging leads to higher Collaborating Causal cluster could be done many ways. Hence this decision making function is passed into Hera as shouldMerge. The final clusters are ranked based on their Collaborating Causality in descending order. This ranking also could be done in multiple ways. Hence ranking function also passed in as parameter *rank*.

Line 2 creates the set of units U observed in Spectra. At lines 3-5, it iterates over U to create initial clusters. Line 4 puts each unit u_i in its own cluster i. Maintaining this bi-map is useful rather than a single map when retrieving the cluster of a unit u_i as shown in getCluster() (lines 15-17). Now the initial clusters should be merged if merging shows stronger collaborating causality than individual clusters. To do that, line 6 gets unit pairs observed in spectra. The pairs are ordered based on the pairwise IonicCL from most likely to least likely (line 7). Lines 8-12 show the iterative process of merging. Merging starts from the most likely pair from orderedPairs. For each pair (u_i, u_j), the clusters containing units u_i and u_j are retrieved using getCluster(). Then line 11 checks whether these two clusters

should be merged according to the shouldMerge() function. If shouldMerge() returns true, the clusters are merged, otherwise original clusters are retained. Since this merging process starts merging based only on pairwise causality instead of considering all collaborating causality information, this is a greedy approach. Once clustering is done, the final clusters are ordered by the rank() function at line 13. Thus, line 13 creates the ranked list of clusters where the cluster with highest collaborating causal likelihood is ranked first. This is our goal. Since the shouldMerge decision and ranking could be done multiple ways. Next, we describe those policies .

Merge policies

The aim of a merge policy is to check whether merging two clusters can create a new potential cluster, with higher collaborating causality than their originals. IonicCL can precisely performs this check. To explore its effectiveness we consider two other merge policies. Both approximate IonicCL with one being a naive baseline. Algorithm 2 shows these merge policies and we describe them in detail as below.

All merge policy functions need to measure causality, whether it is precise or an approximation. To measure causality, we use causal likelihoods calculated by IonicCL formula. This formula can calculate collaborating causal likelihood of a set of units. If the set only contains a single unit, this falls back to returning the causal likelihood of that single unit. getIonicCL() implements the IonicCL equation.

1. Merge_{Ionic}
   This policy checks whether the collaborating causality in a potential new cluster (resulting cluster of merging the two clusters of units u_i and u_j) is higher than the collaborating causality of two individual clusters of units u_i and u_j. All three cluster causal likelihoods are calculated using getIonicCL() at lines 8-11. Next, it compares whether new cluster's collaborating causal likelihood is stronger than both individual clusters' causal likelihoods. If new cluster's collaborating causality is stronger, the policy returns the clusters belonging u_i and u_j should be merged. Since this returns that the most precise collaboration information about a cluster, we name this policy Merge_{Ionic}.

2. Merge_{Approx}
   This policy checks the same thing as policy Merge_{Ionic}, but with an approximation. The geometric mean of a cluster's pairwise collaborating causal likelihoods is used as an approximation to the causal likelihood of the cluster. Function getGeometricMeanofPairCLs() at lines 21-23 shows the geometric mean calculation. First it gets all unit pairs of the cluster. Then, for each pair, the causal likelihood is retrieved using getIonicCL. Finally, the function returns the geometric mean of pairwise causal

likelihoods. Lines 15-18 gets the geometric means and compare them. If, the potential new cluster shows a higher likelihood, policy returns that the clusters belonging u_i and u_j should be merged. Compared to Merge$_{Ionic}$, this policy uses less collaborating causality information, because this is just an approximation done using pairwise collaborating causalities. Hence, we name the policy Merge$_{Approx}$.

3. Merge$_{Naive}$
   This policy checks whether the unit pair shows higher collaboration causality than the individual unit causalities. getIonicCL() is used to get the individual causal likelihoods at lines 2 and 3. The pairwise causal likelihood of pair u_i and u_j is retrieved at line 4. Next, it checks whether the pairwise causal likelihood is stronger than both individual causal likelihoods. If the pairwise collaboration is stronger, the policy returns that the clusters belonging u_i and u_j should be merged. Compared to other two polices, this only considers the pair's collaboration. Since this is a local collaboration check ignoring total cluster likelihoods, we name the policy Merge$_{Naive}$

The amount of collaboration information considered in policies Merge$_{Ionic}$, Merge$_{Approx}$, Merge$_{Naive}$ decreases in order.

Rank policies

The aim of a rank policy is to order a list of clusters by suspiciousness. The first in the list should be the most likely to collaborate to cause the failure. The last in the list should be the least likely to collaborate to cause the failure. To do that, we need to score each cluster numerically. This will represent its Causal Collaboration. We explore two rank policies: one using IonicCL, and the other using the geometric mean of a cluster's pairwise collaborations. We name them Rank$_{Ionic}$ and Rank$_{Approx}$ following the naming pattern we used in merge policies. When multiple clusters have the same causal collaboration, we use standard ranking where ties share what would have been their first ordinal number to rank.

Hera Techniques

Note we have different merge and rank policies. Any merge policy can be combined with any rank policy. We consider each unique combination a Hera technique. As an example, the Hera technique Merge$_{Approx}$-Rank$_{Ionic}$ stands for merging clusters using policy Merge$_{Naive}$ and ranking them based on policy Rank$_{Ionic}$. Suppose S contains the spectra for a program and a test suite, then the invocation of technique Merge$_{Naive}$-Rank$_{Ionic}$ would be Hera(S,Merge$_{Naive}$,Rank$_{Ionic}$). We evaluated all techniques except Merge$_{Ionic}$-Rank$_{Approx}$ as the results of all other 5 techniques showed that Rank$_{Ionic}$ is most successful than Rank$_{Approx}$.

Algorithm 1 Algorithm to create and rank clusters

```
procedure HERA(spectra, shouldMerge, rank)
    U = getUnitSet(spectra)
    for u_i ∈ U do
        cluster[i] = [u_i]
        clusterIDOf[u_i] = i
    pairs = getUnitPairs(spectra)
    orderedPairs = orderByIonicCL(pairs)
    for (u_i, u_j) ∈ orderedPairs do
        cluster_i = getCluster(u_i)
        cluster_j = getCluster(u_j)
        if shouldMerge(u_i, u_j, cluster_i, cluster_j) then
            merge(cluster_i,cluster_j)
    rankedListOfClusters = sort(clusters,rank)

procedure GETCLUSTER(u_i)
    clusterID = clusterIDOf[u_i]
    return cluster[clusterID]
```

Algorithm 2 Algorithms to merge policies

```
procedure MERGE_Ionic(u_i, u_j, cluster_i, cluster_j)
    CL_cluster_i = getIonicCL(cluster_i)
    CL_cluster_j = getIonicCL(cluster_j)
    CL_cluster_i,j = addClusters(cluster_i,cluster_j)
    CL_cluster_i,j = getIonicCL(cluster_i,j)
    return (CL_cluster_i,j >= CL_cluster_i and CL_cluster_i,j >= CL_cluster_j)

procedure MERGE_Approx(l_i, l_j, cluster_i, cluster_j)
    GM_cluster_i = getGeometricMeanOfPairCLs(cluster_i)
    GM_cluster_j = getGeometricMeanOfPairCLs(cluster_j)
    GM_i,j = addClusters(cluster_i,cluster_j)
    GM_cluster_i,j = getGeometricMeanOfPairCLs(cluster_i,j)
    return (GM_cluster_i,j >= GM_cluster_i and GM_cluster_i,j >= GM_cluster_j)

procedure MERGE_Naive(u_i, u_j, cluster_i, cluster_j)
    CL_u_i = getIonicCL({u_i})
    CL_u_j = getIonicCL({u_j})
    CL_u_i,j = getIonicCL({u_i, u_j})
    return (CL_u_i,j >= CL_u_i and CL_u_i,j >= CL_u_j)

procedure GETGEOMETRICMEANOFPAIRCLS(cluster_i)
    P = getUnitPairs(cluster_i)
    return ∛(∏_{{u_i,u_j}∈P} getIonicCL({u_i, u_j}))
```

Motivating example on Hera

The final cluster list created by Hera technique - Merge$_{Approx}$-Rank$_{Ionic}$ for the motivating example is in Figure 3.2 .

Unit Clusters	Rank
4,8	1
2	2
6	3

Figure 3.3: Ranked list for unit clusters Bar.java in Figure 2.1 using Hera. Standard ranking is used to rank the clusters

The ranked list of clusters in Figure 3.3 ranks the faulty lines {4,8} in one cluster at rank 1. And the other units/clusters are ranked below. This achieves our goal for the motivating example. This was not possible with other existing unit level FLTs.

Tractability

Note, units are greedily clustered Hera. This is opposed to finding the optimal partitioning by trying all possible unit combinations as we discussed in section 3.3.1. Suppose a function Foo has |n| units. Next we compare the complexities of Hera and the all combination approach.

In the all combinations approach, the size of the combination list is O(2^n). One IonicCL calculation for each combination is required. Ionic depends on the number of test cases and number of units. The maximum number of test cases is k and maximum number of units is n. Hence, the total complexity is O(2^n * kn).

Hera first creates all unit pairs. The size of this list is O(n^2). Then calculates pairwise IonicCLs initially. We calculated IonicCL complexity as O(kn). Hence the calculating pairwise IonicCLs is O(n^2 * 2k). Afterwards, for each pair, merging does one IonicCL calculation, which is O(kn). Complexity of merging also depends on number of units, hence O(n). Thus, total complexity is O(n^2 (kn + kn + n)) = O(n^3).

This shows that Hera is a tractable and an efficient approach compared to creating a ranked list for all possible combinations.

Chapter 4

Evaluation

We first introduced IonicCL, a formula to calculate collaborating causality of a set of units. Then we presented Hera, an FLT that uses Ionic to partition a set of units and rank them based on their collaborating causality. In this section we evaluate Hera's fault localization capabilities. First we describe our research questions, benchmarks and metrics. Then we present the results for the experiments and answer the research questions.

When we presented Ionic and Hera, we framed them on a 'unit' basis. These experiments are done based on a source code line basis, where every unit is a unique line of source code.

4.1 Research questions

Our research questions as follows. We intend to answer them based on the evaluations.
RQ1. Which Hera technique performs the best in localizing faulty units ?
RQ2: Is Hera capable of localizing better than existing FLTs?
RQ3: Is Hera capable of localizing collaborating causal units better than existing FLTs?

4.2 Benchmarks

We use Defects4J (version 1.0.1) [10] benchmarks for evaluation. Defects4J contains 395 reproducible real world defective versions (programs) from 6 open source Java projects: JFreeChart, Google Closure Compiler, Apache Commons Lang, Apache Commons Math, Mockito and Joda-Time. Table 4.1 shows the number of versions and average lines of code per each defective version in a project.

```
267. public double getNumericalMean() {
268.-    return (double) (getSampleSize() *
        getNumberOfSuccesses()) / (double) getPopulationSize();
269.+    return (getSampleSize() *
        getNumberOfSuccesses()) / (double) getPopulationSize();
269. }
```

Faulty unit	Fault type	Candidate units
268	Incorrect	268

Figure 4.1: Patch extract and ground truth table for Math-2 from Defects4J

Project	#Versions	LoC
Chart	26	132.2k
Mockito	38	103.9k
Closure	133	216.2k
Lang	65	49.9k
Math	106	103.9k
Time	27	105.2k
Total	395	

Table 4.1: Defects4J (version 1.1.0) and number of versions and LoC information. 'LoC' is average lines of code for each version version of the project [21] .

4.2.1 Definitions

Defects4J provides several types of information. We here introduce definitions related to Defects4J.

Defect

A defect is a flaw (undesired behaviour) in the source code. Each version has one or more defects co-existing with each other.

Faulty unit

This is a faulty element in the source code. A defect is caused by one or more faulty units. Every Defects4J faulty unit has a faulty type. There are two faulty types, *Incorrect* and *Omitted*. Incorrect ones need to be modified to be fixed. Omitted ones need new code to be inserted to be fixed. Figures 4.1 and 4.2 and show patch extracts and ground truth tables for two Defects4J versions, Math-2 and Closure-15. Math-2 has one incorrect type faulty unit and Closure-15 has one omitted type faulty unit. Each version has a set of passing and failing test cases. Failing test cases reveal at least one of these faulty units.

```
84. public boolean apply(Node n) {
...
94.     if (n.isCall() && NodeUtil.functionCallHasSideEffects(n)) {
95.         return true;
96.     }
97.
98.     if (n.isNew() && NodeUtil.constructorCallHasSideEffects(n)) {
99.         return true;
100.    }
101.
102.+   if (n.isDelProp()) {
103.+       return true;
104.+   }
...  }
```

Faulty unit	Fault type	Candidate units
102	Omitted	103, 94, 98

Figure 4.2: Patch extract and ground truth table for Closure-15 from Defects4J

Candidate unit

The location of faulty unit might not always be the location to fix a bug. Recall, there could be two types of faulty units, omitted and incorrect. For incorrect units, the fix location is the faulty unit location itself. Omitted units might have more than one suitable location to be inserted. In the example code patch from Figure 4.2, the missing if condition could have been added at lines 94, 98 or 103. All locations are correct. Therefore, Defects4J defines candidate units. A candidate unit is the code location at which a code block is modified or inserted to fix a faulty unit. Note the candidate units for each faulty unit in Figures 4.2 and 4.1. We consider a faulty unit is localized as soon as any candidate unit is reached in the ranked list.

4.2.2 Limitations

Non-executable code locations

Programs has non-executable pieces of code such as declarations. Defects4J has some defective versions with faulty units involving only non-executable code. As Ionic requires a unit to be executed in order to find its causality, these versions do not qualify as valid for our evaluations. Out of 395 versions, 383 versions were valid.

Lack of collaborating causality information

Our third research question intends to find how efficient Ionic is at localizing defects with collaborating faulty units compared to existing FLTs. For this research question, we would like to evaluate how Ionic would perform on versions with multiple collaborating faulty units. However, Defects4J doesn't provide collaborating causality information of versions. The best alternative is to select the versions with multiple faulty units. They have at least a chance of collaborating to cause the failures, but a version with multiple faulty units might not always mean the faulty units must collaborate to cause the failures. This is explained using the example in Figure 4.2. Suppose there is another similar omitted condition. Defects4J considers this as a second faulty unit in the version. Candidate units for the second faulty unit also would be the same candidate units, 103, 94, 98. This is correct because both could be inserted into the same locations. When any of these candidate units is reached in the ranked list, both faulty units are localized. This shows that, though multiple missing code pieces gets inserted to separate locations in bug fixing, predicting where they could be inserted does not necessarily lead to multiple locations. This limitation can only be overcome by providing more information about the ground truth. This requires human assistance. For our evaluation purposes we assume a version's k faulty units will always be localized by k unique candidate units. Thus, there is a chance of collaborating causality when k>1. Figure 4.3 shows the number of faulty unit counts in versions.

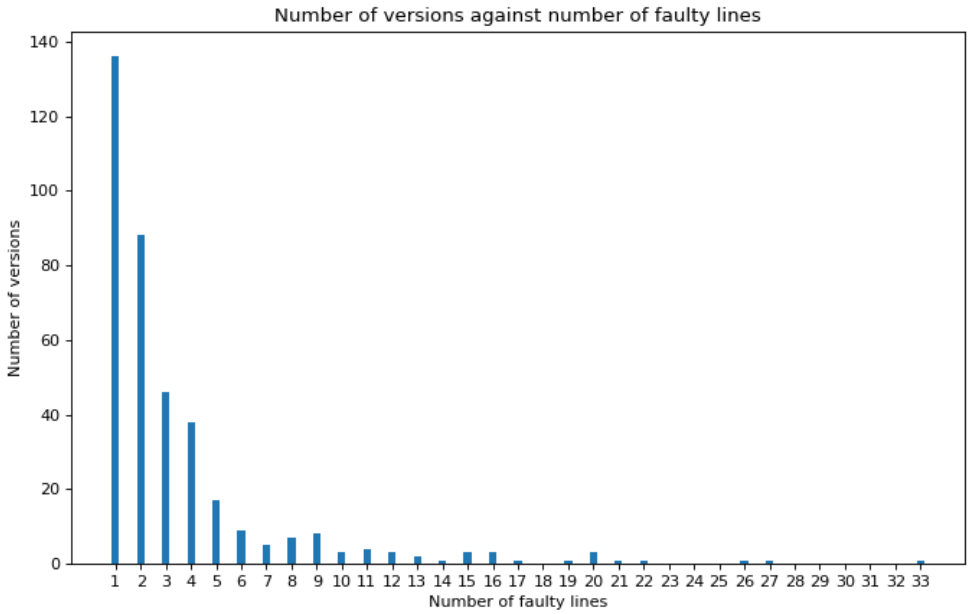


Figure 4.3: Number of versions against number of faulty units. Graph is read as, x faulty units are present in y versions

4.3 Metrics

To answer our research questions we have to compare Hera techniques against each other and against existing FLTs. We introduce 6 metrics to measure the quality of a FLT.

Hera's output is a ranked list. A developer examines clusters according to the list to localize faults. Orso and Parnin's work [15] presents a study on the number ranked items a developer is willing to examine from a ranked list. Based on this work we set the upper bound on the number of examined clustered to 10. Doric [12] also uses the same upper bound.

Some metrics we present measure cluster quality while others measure unit quality. We use cluster metrics when comparing Hera techniques against each other. We use unit metrics when we compare Ionic with SBFL and Doric.

4.3.1 k@n

$k@n_{cluster-n}$

k@n for cluster n is the number of the versions that localized k faulty units within the top n *clusters* in the ranked list. The higher the score, the better the technique.

$k@n_{unit-n}$

k@n for unit n is the number of the versions that localized k faulty units within the top n *units* in the ranked list. The higher the score, the better the technique.

We explore k@n metrics only when k $\in$ {1,2,3}. Suppose k = 3, it evaluates the instance of localizing the *3*rd faulty unit. This should be evaluated on the versions with at least 3 faulty units. Figure 4.3 shows there are only 159 versions falls under this condition. When k >= 4 this number drops below 90(<25% of the benchmarks). An evaluation on such a small sample may not provide enough evidence to draw conclusions with high confidence.

4.3.2 Recall

The number of successfully retrieved relevant items out of all relevant items. The higher the recall, the better the technique.

Cluster Recall

Recall for cluster n is the number of localized faulty units within the top *n clusters* in the ranked list out of all faulty units.

$$Recall_{cluster-n} = \frac{\sum_n FaultyUnits_{cluster-n}}{\sum FaultyUnits} \tag{4.1}$$

Unit Recall

Recall for unit n is the number of localized faulty units within the top *n units* in the ranked list out of all faulty units.

$$Recall_{unit-n} = \frac{\sum_n FaultyUnits_{unit-n}}{\sum FaultyUnits} \tag{4.2}$$

4.3.3 Precision

The number of successfully retrieved relevant items out of all retrieved items. The higher the precision, the better the technique. Note, a *faulty cluster* is a cluster carrying at least one faulty unit.

Cluster Precision

Precision for cluster n is the fraction of *faulty clusters* within the top *n clusters* in the ranked list.

$$Precision_{cluster-n} = \frac{\sum_n FaultyClusters_{cluster-n}}{n} \tag{4.3}$$

Intra-cluster Precison

Intra-cluster precision for cluster n is the number of faulty units within the top *n faulty clusters* in the ranked list out of all units in *those faulty clusters*.

$$Intra - clusterPrecision_{cluster-n} = \frac{\sum_n FaultyUnits_{cluster-n}}{\sum Units_{FaultyCluster_n}} \tag{4.4}$$

Cluster precision captures how many clusters were buggy out of n clusters retrieved. Intra-cluster precision (4.4) measures the portion of noise (non-faulty) units in returned faulty clusters.

Unit Precision

Precision for unit n is the fraction of faulty units within the top *n units* in the ranked list.

$$Precision_{unit-n} = \frac{\sum_n FaultyUnits_{unit-n}}{n} \tag{4.5}$$

We presented two cluster related precision metrics. But we could give only one interpretation for cluster recall. Recall measures the portion retrieved out of relevant things. Defects4J benchmarks only provides ground truth about relevant things in terms of units. We do not have information on how many clusters they should belong to. Suppose a version has three faulty units, and two of them show collaborating causality. Then the ground truth should

specify those two faulty units should be in the same cluster. Human assistance is needed to enhance the information provided by ground truth.

4.4 Setup

We implemented a statement profiling algorithm for Java programs [3] using Soot framework[19]. We then implemented the Hera along with its merge and rank policies. For each version of Defects4J, we instrumented the source code, ran the test cases, and collected the profiles. We created the spectra and coverage matrix using the profiles obtained. We then used spectra and coverage matrix to produce the ranked list of clusters using Hera.

4.5 Evaluating Hera techniques

In this section we answer our first research question, 'RQ1: Which Hera technique performs the best in localizing faulty units?'. In this question, we need to compare Hera techniques. Therefore, we use results of cluster related metrics ($\text{Recall}_{\text{cluster-n}}$, $\text{Precision}_{\text{cluster-n}}$, Intra-$\text{ClusterPrecision}_{\text{cluster-n}}$ and $\text{k@n}_{\text{cluster-n}}$) and cluster sizes to answer this question. We first present the results and then analyse and answer the research question.

4.5.1 Analysis

Figure 4.4 shows the results for the metric $\text{Recall}_{\text{cluster-n}}$. Intuitively this captures the rate of localizing faulty units by the benchmarks. The bottom 25% of versions behave the same way for every technique. But the other trend lines show, technique Merge_{Ionic}-Rank_{Ionic} has the best scores. 50% of versions of Merge_{Ionic}-Rank_{Ionic} achieve 20% recall at the 3rd cluster. Afterwards, it increases and reaches to 68% by the 10th cluster. Whereas, 50% was any other technique's best. Comparing the top 25% versions trend lines shows that Merge_{Ionic}-Rank_{Ionic} is the first to achieve 100% recall.

Figure 4.5 shows the results for metric $\text{Precision}_{\text{cluster-n}}$. This captures how many clusters contained faulty unit information within top n clusters of the ranked list. Again, technique Merge_{Ionic}-Rank_{Ionic} stands out as half of its versions achieved 33% at the 3rd cluster. The second best technique, Merge_{Naive}-Rank_{Ionic} only achieved 25% at its best. Top 25% line of techniques using Merge_{Approx} policy starts at 50% precision. This shows that these techniques struggle localizing a faulty unit at its first cluster.

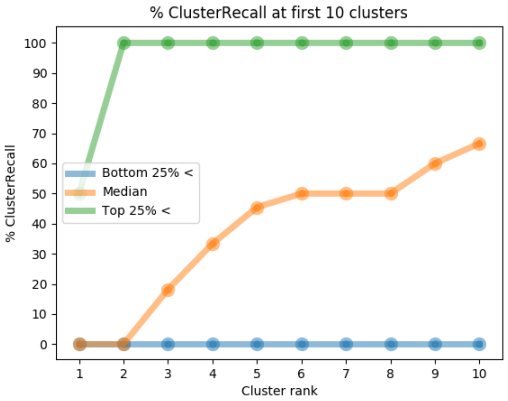
Table 4.3 shows results for $\text{k@n}_{\text{cluster-n}}$ metric. This metric counts the number of versions that could localize the first, second, and third faulty units and present them as a percentage of the number of versions it was evaluated on. Again, technique Merge_{Ionic}-Rank_{Ionic} does the best by achieving 65% , 54%, 49% beating any other technique in localizing first three faulty units. This matches to the results interpretation of cluster recall in Figure 4.4.

Table 4.2 shows the results for metric Intra-ClusterPrecision$_{\text{cluster-n}}$. This represents noise (non-faulty units), carried by clusters carrying faulty units within top n clusters. By the 10th cluster, techniques using Merge$_{Approx}$ reach 52%. This conveys that, half the units carried by faulty clusters is found to be faulty. This is the highest any technique has achieved. Technique Merge$_{Ionic}$-Rank$_{Ionic}$, which had been performing best so far comes in last.

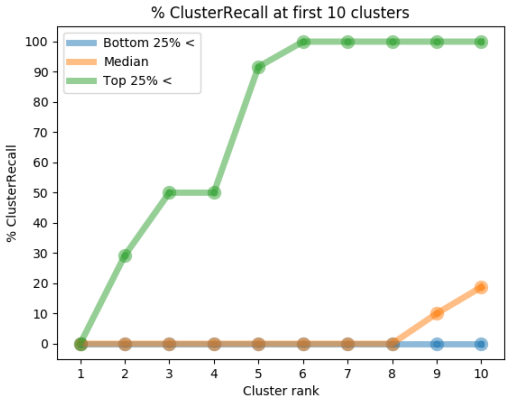
RQ1

Technique Merge$_{Ionic}$-Rank$_{Ionic}$, is best at localizing faulty units and ranking faulty clusters but has more noise in clusters comparatively as seen in the intra-cluster precision. Therefore, if a user is interested in localizing collaborating faulty units highly while ignoring the noise content, this might be a good technique to use.

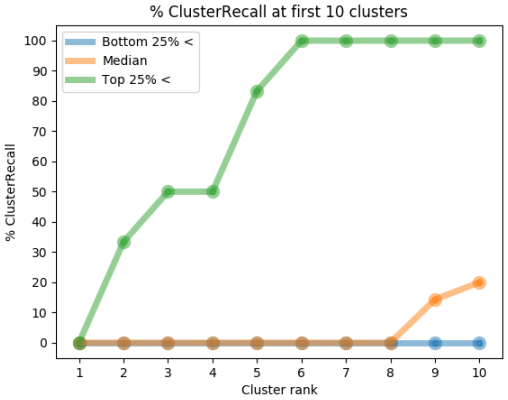
Techniques using Merge$_{Approx}$ policy perform poorly in localizing faulty units but shows better intra-cluster precision. Table cluster sizes at 4.4 shows that their clusters are smaller compared to Merge$_{Ionic}$-Rank$_{Ionic}$ technique. This may be a reason for low recall and high intra-cluser precision in those techniques. These would be good techniques for users like developers who care about the noise content within the clusters.



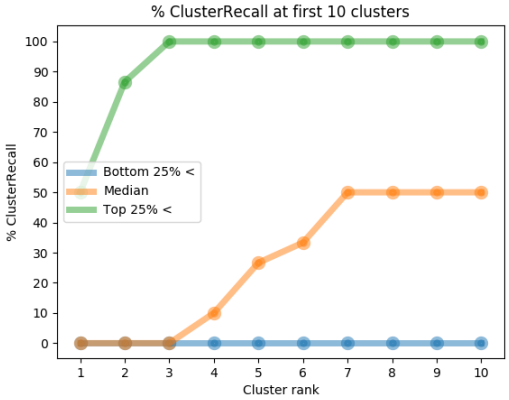
(a) Merge$_{Ionic}$-Rank$_{Ionic}$



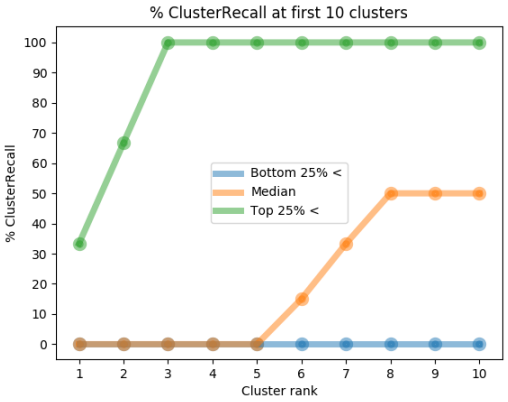
(b) Merge$_{Approx}$-Rank$_{Ionic}$



(c) Merge$_{Approx}$-Rank$_{Approx}$

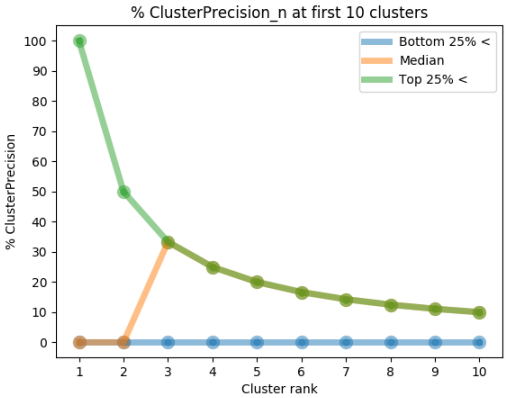


(d) Merge$_{Naive}$-Rank$_{Ionic}$

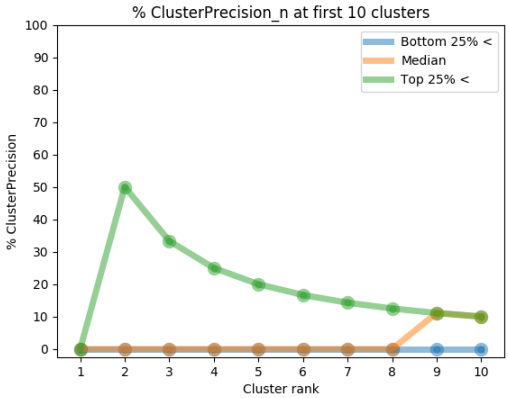


(e) Merge$_{Naive}$-Rank$_{Approx}$

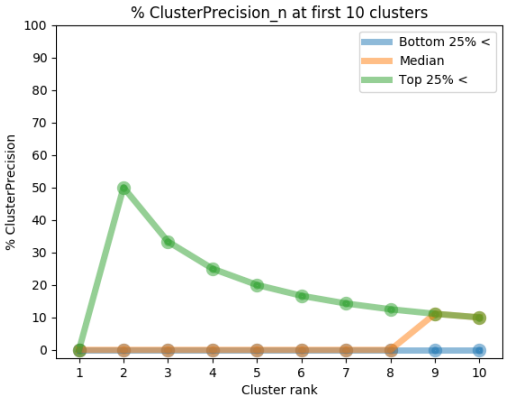
Figure 4.4: Recall$_{\text{cluster-n}}$: Cluster recall as a percentage against cluster rank for n $\in$ range [1..10]. Each graph shows one Hera technique. Each point is read as "At cluster x, y% of faulty units localized". The orange trend line shows the behaviour of the median of the versions. Likewise, blue and green show the behaviour of bottom and top 25% respectively.



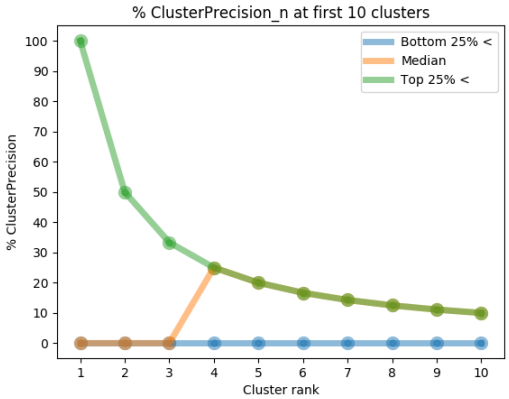
(a) Merge$_{Ionic}$-Rank$_{Ionic}$



(b) Merge$_{Approx}$-Rank$_{Ionic}$



(c) Merge$_{Approx}$-Rank$_{Approx}$



(d) Merge$_{Naive}$-Rank$_{Ionic}$

(e) Merge$_{Naive}$-Rank$_{Approx}$

Figure 4.5: Precision$_{\text{cluster-n}}$: Cluster precision as a percentage against cluster rank for n $\in$ range [1..10]. Each graph shows one Hera technique. Each point is read as "At cluster x, y% of clusters carries faulty unit s in them". The orange trend line shows the behaviour of the median of the versions. Likewise, blue and green show the behaviour of bottom and top 25% respectively.

Technique	#versions	k	@n			
			@1(%)	@3(%)	@5(%)	@10(%)
Merge_{Ionic}-Rank_{Ionic}	383	1	32	51	57	65
Merge_{Approx}-Rank_{Ionic}	383	1	20	36	44	52
Merge_{Approx}-Rank_{Approx}	383	1	21	37	44	52
Merge_{Naive}-Rank_{Ionic}	383	1	31	46	53	60
Merge_{Naive}-Rank_{Approx}	383	1	28	44	49	58
Merge_{Ionic}-Rank_{Ionic}	247	2	23	40	47	54
Merge_{Approx}-Rank_{Ionic}	247	2	13	23	31	39
Merge_{Approx}-Rank_{Approx}	247	2	14	23	31	40
Merge_{Naive}-Rank_{Ionic}	247	2	19	36	42	47
Merge_{Naive}-Rank_{Approx}	247	2	18	33	38	45
Merge_{Ionic}-Rank_{Ionic}	159	3	17	32	40	49
Merge_{Approx}-Rank_{Ionic}	159	3	6	13	22	35
Merge_{Approx}-Rank_{Approx}	159	3	7	14	23	35
Merge_{Naive}-Rank_{Ionic}	159	3	13	29	36	43
Merge_{Naive}-Rank_{Approx}	159	3	11	26	31	39

Table 4.3: $\text{k@n}_{\text{cluster-n}}$: k@n results as a percentage, when k $\in$ {1,2,3} and n $\in$ {1,3,5,10}. Each numeric cell interpreted as the % of versions localized k faulty units within top n clusters. #versions changes with the k, as we run the metric on versions with at least k faulty units (3.3).

Technique	#versions	@1	@3	@5	@10
Merge_{Ionic}-Rank_{Ionic}	383	23.17	24.11	22.74	23.12
Merge_{Approx}-Rank_{Ionic}	383	53.95	53.79	53.13	52.65
Merge_{Approx}-Rank_{Approx}	383	56.03	53.15	52.88	52.63
Merge_{Naive}-Rank_{Ionic}	383	23.23	24.92	23.71	23.65
Merge_{Naive}-Rank_{Approx}	383	26.59	27.55	26.88	24.37

Table 4.2: $\text{Intra-ClusterPrecision}_{\text{cluster-n}}$: Intra-cluster precision for Hera techniques as a percentage, for n $\in$ {1,3,5,10}. Each numeric cell interpreted as, the geo metric mean of intra-cluster precision for top n clusters

Technique	Cluster size(Geometric mean)	StdDev
Merge_{Ionic}-Rank_{Ionic}	7	11
Merge_{Approx}-Rank_{Ionic}	2	1
Merge_{Approx}-Rank_{Approx}	2	2
Merge_{Naive}-Rank_{Ionic}	6	10
Merge_{Naive}-Rank_{Approx}	5	9

Table 4.4: Geometric mean of cluster sizes for each technique. Values are rounded off to integer as cluster sizes would be in reality.

4.6 Evaluating Hera vs other FLTs

In this section we answer the second and the third research questions.
RQ2: Is Hera capable of localizing faulty units better than existing FLTs?
RQ3: Is Hera capable of localizing collaborating faulty units better than existing FLTs?

Both these questions require comparing Ionic against existing FLTs. We compare Ionic results against Doric and Ochiai and Dstar2 from SBFL family. As per related work [21], Ochiai and Dstar2 showed better localization capabilities compared to other SBFL technique on Defects4J benchmarks. These existing FLTs produce the the ranked list in units. So, we use unit related metrics($\text{Recall}_{\text{unit-n}}$, $\text{Precision}_{\text{unit-n}}$, $\text{k@n}_{\text{unit-n}}$) to do this evaluation. First we describe how we obtained results of other FLTs. Hera outputs a cluster ranked list. It needs to be converted to a unit ranked list to compare with other FLTs. Therefore, we describe how we convert a Hera ranked list of clusters to a ranked list of units. Then we present results, analysis and answer both research questions.

4.6.1 Obtaining other FLTs' results

We use scripts provided in Evaluating FLT [16] to create ranked list of units for SBFL techniques. We use IonicCL to obtain Doric's ranked list of units. Note that IonicCL falls back to Doric's causal likelihood calculation when the set in consideration is just one unit.

4.6.2 Converting Hera's ranked list of clusters to a ranked list of units

Hera outputs a ranked list of unit clusters. But for unit related metrics, we need to retrieve the top n *unit* related results. We create a ranked list of units starting from the 1st ranked cluster. Ordering of the units from the same cluster is considered in three cases.
Best case: All faulty units in cluster are examined at the beginning of each cluster. Non-faulty units are examined at last.
Average case : Half of the non-faulty units in the cluster examined at first. Then faulty units are examined. The second half of the non-faulty units are then examined.
Worst case : All non-faulty units in the cluster are examined at the beginning of each clus-

ter. Then faulty units are examined.

If multiple clusters have the same rank, ordering of clusters is also based on above three cases. Such as, best case being the instance where all clusters carrying faulty units examined first and followed by other non-faulty unit clusters. So, this leads to 9 possible ways of creating ranked unit list if there are tied clusters.

Consider the cluster ranked list in 4.6 and relevant faulty unit information. Figure 4.7 shows unit ranked lists for the three cases. In our evaluation we only present the results for average case ordering of both clusters and units.

Rank	Cluster	Faulty units localized
1	21, 22, 23, 24	21, 24
2	27, 28, 30	28
3	32	30

Figure 4.6: Ranked list of clusters and faulty unit

Rank	BestCase	AverageCase	WorstCase
1	21	22	22
2	24	21	23
3	22	24	21
4	23	23	24
5	28	27	27
6	27	28	30
7	30	30	28
8	32	32	32

Figure 4.7: Ranked list of clusters converted to units according to the three cases

4.6.3 Results

Results for $\text{Recall}_{\text{unit-n}}$ and $\text{Precision}_{\text{unit-n}}$ (Figures 4.8, 4.9) contain Hera techniques and Doric, but not Ochiai and Dstar. These graphs need results at each nth unit. We use scripts from [16] to calculate Ochiai and Dstar results, and they don't provide results at each unit n. Results for $\text{k@n}_{\text{unit-n}}$ (Table 4.5) contains Dstar2 and Ochiai results when k=1, but not when k $\in \{2, 3\}$. This is due to the same limitations in scripts by [16]. They only provide the ranks of first, last and a median faulty units. Localizing the first faulty unit is equivalent to k=1 in $\text{k@n}_{\text{unit-n}}$ metric. However, median and last are relative positions in a set of faulty units. Hence, cannot find exact ranks of 2nd and 3rd faulty units. We present

k@$n_{\text{unit-n}}$ results when k $\in$ {median, last} for Ochiai and Dstar. When k $\in$ {median, last}, the number of versions evaluated is 247. When evaluating results for k@$n_{\text{unit-n}}$ we consider versions with at least k faulty units. But median and last cannot be expressed in exact numbers. Therefore, we consider any version with at least 2 faulty units qualify, hence 247 versions. We think presenting Doric results for k@$n_{\text{unit-n}}$ when k $\in$ {1, 2, 3} is much and sufficient. Thus, we do not present Doric when k $\in$ {median, last}

4.6.4 Analysis

Figure 4.8 shows $\text{Recall}_{\text{unit-n}}$ results. Blue and orange trend lines are overlapping at 0% on y axis. Therefore, all techniques perform poorly on the on bottom 50% of versions. Doric's top 25% of versions start localizing faulty units at the 6th unit and rises to 40% by the 10th unit. Merge_{Ionic}-Rank_{Ionic} starts at the 7th unit but reaches 50% eventually and the localization rate is higher than Doric. Techniques using Merge_{Approx} stand out. Naive merging techniques (Merge_{Naive}-Rank_{Ionic}, Merge_{Naive}-Rank_{Approx}) behave similarly to Merge_{Ionic}-Rank_{Ionic}.

Figure 4.9 shows $\text{Precision}_{\text{unit-n}}$ results. This metric captures the non-faulty units within the first n units. Techniques using Merge_{Approx} stand out. All the other techniques perform similarly to each other.

Table 4.5 presents k@$n_{\text{unit-n}}$ results. Techniques using Merge_{Approx} perform better for all k values. When k=1 and n=10, Merge_{Ionic}-Rank_{Ionic} and Naive merging techniques tie with Doric but perform worse than Dstar2 and Ochiai. When k>1 Merge_{Ionic}-Rank_{Ionic} does better than Doric and SBFL techniques.

In all the above results, techniques with Merge_{Approx} shows better results than other Hera techniques. But the difference it shows may not be entirely true in reality. Because we use average case ordering when converting the cluster list to a unit list. In this case, faulty units always considered to be found at the middle of the cluster. Hence, smaller clusters with some faulty units gets a huge advantage compared to large clusters though they have many faulty units. Recall, techniques with Merge_{Approx} have small clusters (Figure 4.4). Thus, they get the edge compared to other techniques' large clusters.

RQ2

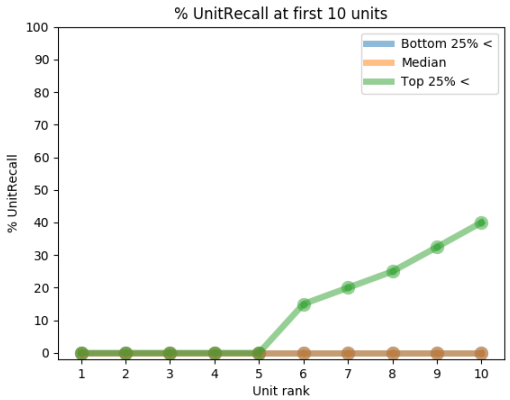
We answer the second research question, "RQ2: Is Ionic capable of localizing better than existing FLTs?". Analysis indicated that all Hera techniques mostly perform better or as well as Doric. The only exceptions inspected at $\text{Recall}_{\text{unit-n}}$ when Doric showed better localization at the 6th cluster. This might be caused by unit ranked list creation limitations we discussed in the analysis. SBFL techniques perform better in localizing first faulty unit than some Hera techniques. When localizing subsequent faulty units, Hera always performs better.

Even when we consider the cases where other existing FLTs have better results compared to Hera, we should not ignore the fact that we might be comparing Hera's one *cluster* result

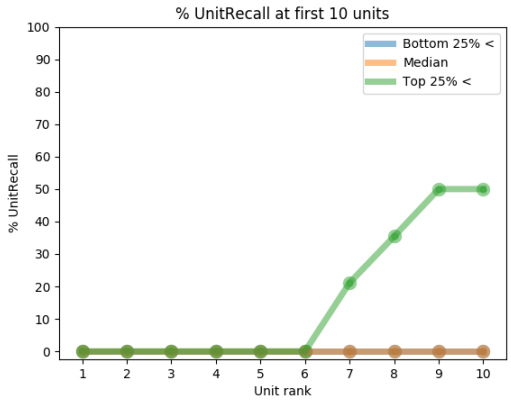
against other FLTs' 10 *units*. Because Figure 4.4 shows that clusters made by Merge_{Approx} and Merge_{Naive} could be large as 6 or 7 *units*. According to Orso and Parnin [15] a developer might be only willing to examine up to 10 ranks in the output. Therefore Hera may have few other clusters a developer is interested to examine while other FLTs have exhausted their lists.

RQ3

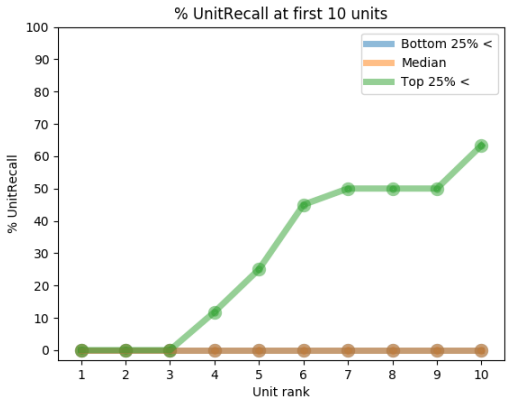
We answer the third research question, "RQ3: Is Hera capable of localizing collaborating faulty units better than existing FLTs?" Since we are focused on *collaborating faulty units*, we base our answer on $k@n_{\text{unit-n}}$ metric when k>1. Analysis showed that all Hera techniques outperformed other FLTs when k>1.



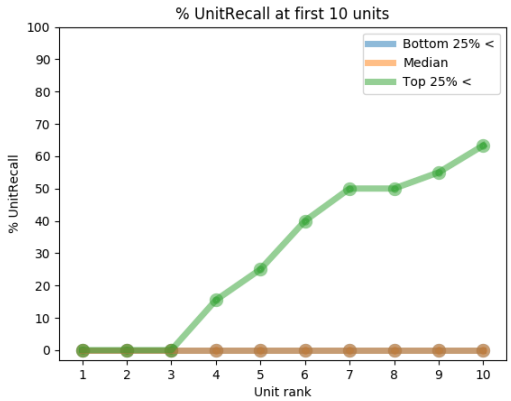
(a) Doric



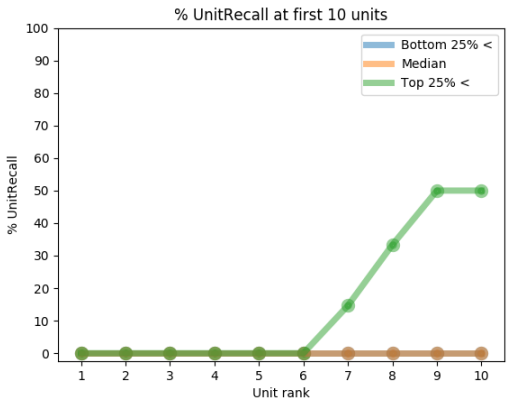
(b) Merge$_{Ionic}$-Rank$_{Ionic}$



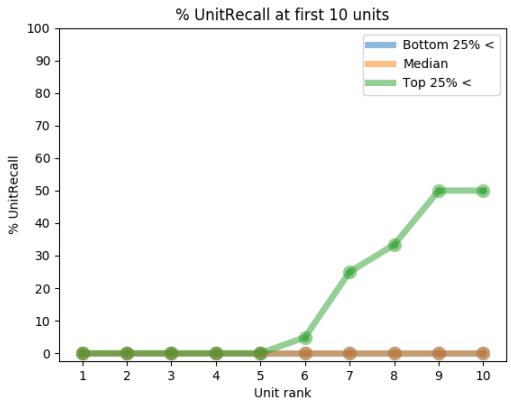
(c) Merge$_{Approx}$-Rank$_{Ionic}$



(d) Merge$_{Approx}$-Rank$_{Approx}$

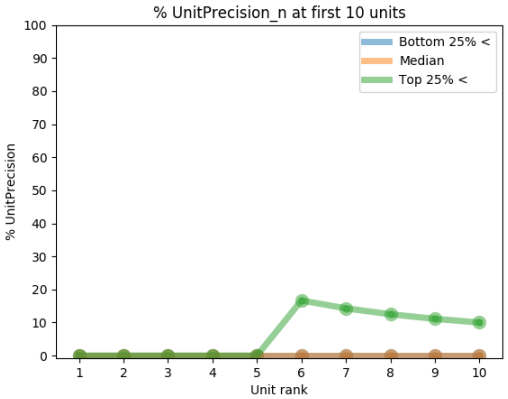


(e) Merge$_{Naive}$-Rank$_{Ionic}$

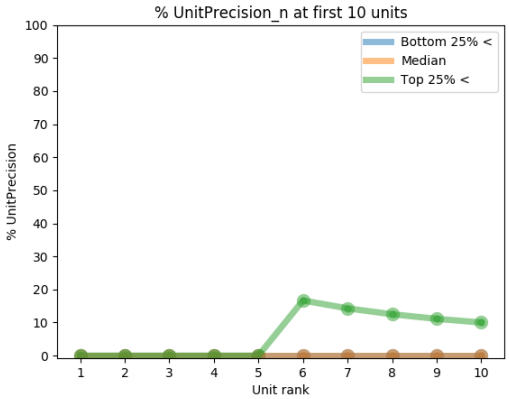


(f) Merge$_{Naive}$-Rank$_{Approx}$

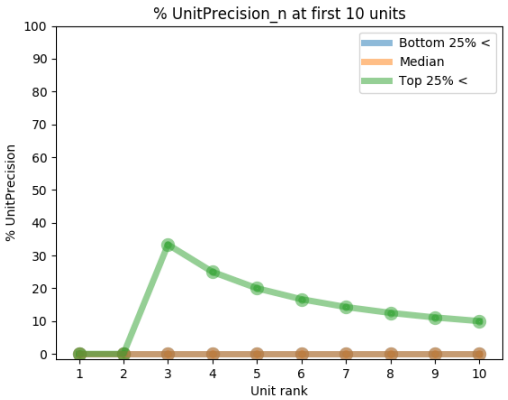
Figure 4.8: Recall$_{\text{unit-n}}$: Unit recall as a percentage against unit rank for n $\in$ range [1..10]. Each graph shows one Hera technique. Each point is read as "At cluster x, y% of faulty units localized". The orange trend line shows the behaviour of the median of the versions. Likewise, blue and green show the behaviour of bottom and top 25% respectively. Blue and orange lines are overlapping in these cases



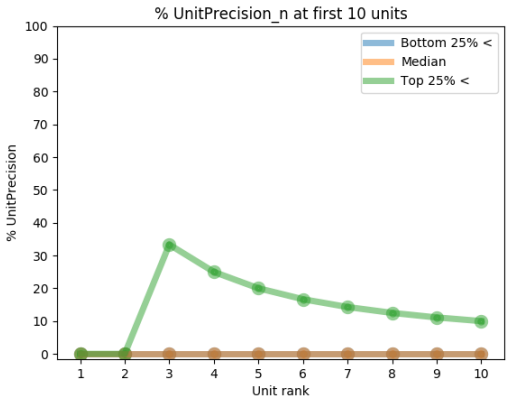
(a) Doric



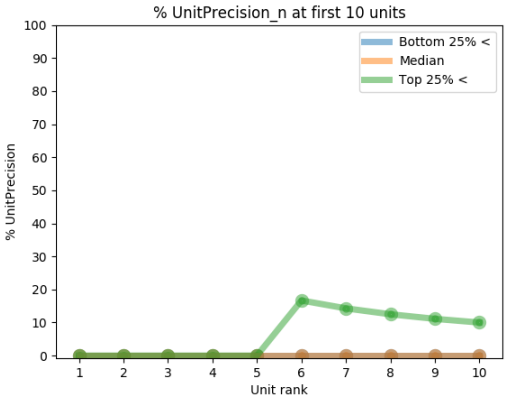
(b) Merge$_{Ionic}$-Rank$_{Ionic}$



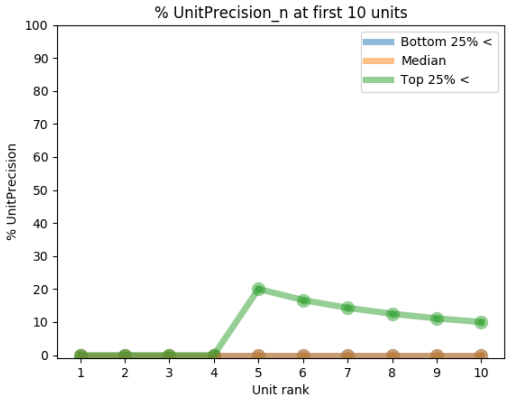
(c) Merge$_{Approx}$-Rank$_{Ionic}$



(d) Merge$_{Approx}$-Rank$_{Approx}$



(e) Merge$_{Naive}$-Rank$_{Ionic}$



(f) Merge$_{Naive}$-Rank$_{Approx}$

Figure 4.9: Precision$_{\text{unit-n}}$: Unit precision as a percentage against unit rank for n $\in$ range [1..10]. Each graph shows one Hera technique. Each point is read as "At cluster x, y% of faulty units localized". The orange trend line shows the behaviour of the median of the versions. Likewise, blue and green show the behaviour of bottom and top 25% respectively. Blue and orange lines are overlapping in these cases

Technique	#versions	k	@n			
			@1(%)	@3(%)	@5(%)	@10(%)
Merge_{Ionic}-Rank_{Ionic}	383	1	0	11	20	36
Merge_{Approx}-Rank_{Ionic}	383	1	0	23	30	43
Merge_{Approx}-Rank_{Approx}	383	1	0	22	31	43
Merge_{Naive}-Rank_{Ionic}	383	1	0	11	19	36
Merge_{Naive}-Rank_{Approx}	383	1	0	12	21	36
Doric	383	1	0	14	23	36
Dstar2	383	1	5	19	28	38
Ochiai	383	1	4	20	30	39
Merge_{Ionic}-Rank_{Ionic}	247	2	0	1	9	23
Merge_{Approx}-Rank_{Ionic}	247	2	0	6	18	28
Merge_{Approx}-Rank_{Approx}	247	2	0	6	18	29
Merge_{Naive}-Rank_{Ionic}	247	2	0	1	8	22
Merge_{Naive}-Rank_{Approx}	247	2	0	1	10	24
Doric	247	2	0	4	13	22
Merge_{Ionic}-Rank_{Ionic}	159	3	0	0	2	15
Merge_{Approx}-Rank_{Ionic}	159	3	0	0	8	19
Merge_{Approx}-Rank_{Approx}	159	3	0	0	8	19
Merge_{Naive}-Rank_{Ionic}	159	3	0	0	1	13
Merge_{Naive}-Rank_{Approx}	159	3	0	0	1	15
Doric	159	3	0	0	3	14
Merge_{Ionic}-Rank_{Ionic}	247	Median	0	4	11	23
Merge_{Approx}-Rank_{Ionic}	247	Median	0	9	15	25
Merge_{Approx}-Rank_{Approx}	247	Median	0	9	15	26
Merge_{Naive}-Rank_{Ionic}	247	Median	0	4	10	22
Merge_{Naive}-Rank_{Approx}	247	Median	0	4	11	23
Dstar2	247	Median	2	8	13	20
Ochiai	247	Median	2	9	13	19
Merge_{Ionic}-Rank_{Ionic}	247	Last	0	4	9	19
Merge_{Approx}-Rank_{Ionic}	247	Last	0	9	13	21
Merge_{Approx}-Rank_{Approx}	247	Last	0	9	13	22
Merge_{Naive}-Rank_{Ionic}	247	Last	0	4	8	19

Continued on next page

Table 4.5 – *Continued from previous page*

Technique	#versions	k	@n			
			@1(%)	@3(%)	@5(%)	@10(%)
Merge$_{Naive}$-Rank$_{Approx}$	247	Last	0	4	9	19
Dstar2	247	Last	2	6	10	14
Ochiai	247	Last	1	7	11	15

Table 4.5: k@n$_{\text{unit-n}}$: k@n results as a percentage, when k $\in$ {1, 2, 3, median, last} and n $\in$ {1,3,5,10}. Each numeric cell is interpreted as the % of versions localized k faulty units within top n clusters. #versions change with the k.

4.7 Case Studies

In this section we present two case studies to reason some trends we examined.

4.7.1 Merge$_{Ionic}$-Rank$_{Ionic}$ vs Merge$_{Naive}$-Rank$_{Ionic}$

We present this case study to show why Merge$_{Ionic}$-Rank$_{Ionic}$ performs better compared to Merge$_{Naive}$-Rank$_{Ionic}$. Consider the source code from chart-25 from Defects4J (4.10) in which line 402 is the faulty unit. The ranked list of clusters using Merge$_{Ionic}$-Rank$_{Ionic}$ and Merge$_{Naive}$-Rank$_{Ionic}$ techniques are in Figures 4.11, 4.12 respectively. They show Merge$_{Ionic}$-Rank$_{Ionic}$ ranked the cluster containing faulty unit at rank 1 while Merge$_{Naive}$-Rank$_{Ionic}$ ranked it last. The reason is as follows.

Consider the simplified coverage matrix and ranked unit pairs extract as shown in Figures 4.13 and 4.14. All pairs have the same CL. Note that pair {411,490} is not a valid pair, hence not ranked in pairwise CL calculation. Merge$_{Naive}$-Rank$_{Ionic}$ merges clusters based on pairwise CL and pairwise co-execution. Faulty line 402 co-executed with 411 in first test case. 402 co-executed with 490 in second test case. All pairs have the same CL. Therefore, lines 402, 409 and 411 is clustered together, as they co-executed as pairs in at least one of the test case and carries the same CL. Note that it misses the fact 402, 411 and 490 never co-executed in a single test case. However, note that Merge$_{Naive}$-Rank$_{Ionic}$ ranks using Rank$_{Ionic}$. Rank$_{Ionic}$ uses IonicCL as scoring function. In order to get a positive IonicCL value, all units in a cluster should have been co-executed in a failing test case. Since this cluster doesn't qualify that, it gets 0 and ranked last. In contrast, Merge$_{Ionic}$-Rank$_{Ionic}$ merges using IonicCL itself. Hence, it considers all the connections globally. Thus, it identifies that there is not enough evidence to put 411 and 490 in the same cluster. This leads to two clusters with the same CL. This might be a reason why Merge$_{Ionic}$-Rank$_{Ionic}$ shows

better results than Merge$_{Naive}$-Rank$_{Ionic}$.

```
402.    Number meanValue = dataset.getMeanValue(row, column);
...
410.    if (value >= uclip) {
411.        return;
412.    }
...
490.  if (generator != null && isItemLabelVisible(row, column)) {
...
```

Figure 4.10: Source code drawn from Chart-25 from Defects4J

Cluster	IonicCL	Rank
411	0.001	1
402,410,490	0.001	1

Figure 4.11: Ranked list of clusters for Chart-25 using technique Merge$_{Ionic}$-Rank$_{Ionic}$

Cluster	IonicCL	Rank
402,410,411,490	0	Last

Figure 4.12: Ranked list of clusters for Chart-25 using technique Merge$_{Naive}$-Rank$_{Ionic}$

$$\begin{array}{c} \\ t_1 \\ t_1 \end{array} \begin{array}{c} \begin{array}{ccccc} 402 & 410 & 411 & 490 & e \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

Figure 4.13: Simplified coverage matrix for Chart-25

Pair	IonicCL	Rank
411,411	0.001	1
410,490	0.001	1
410,411	0.001	1
402,490	0.001	1
402,410	0.001	1
490,490	0.001	1
402,402	0.001	1

Figure 4.14: Ranked list of pairs for Chart-25

4.7.2 Merge$_{Ionic}$-Rank$_{Ionic}$ vs Merge$_{Approx}$-Rank$_{Ionic}$ vs Doric

We present this case study to explain the trends of Merge$_{Ionic}$-Rank$_{Ionic}$, Merge$_{Approx}$-Rank$_{Ionic}$ and Doric. Consider the source code from Lang-17 from Defects4J (4.10) in which line 83, 89 and 102 are the faulty units. Ranked list of clusters using Merge$_{Ionic}$-Rank$_{Ionic}$, Merge$_{Approx}$-Rank$_{Ionic}$ and Doric techniques are in Figures 4.16 ,4.17 and 4.18 respectively.

Merge$_{Ionic}$-Rank$_{Ionic}$ is able to cluster the three faulty units into one higher ranked cluster while Merge$_{Approx}$-Rank$_{Ionic}$ separated the faulty units into two clusters. This explains the smaller cluster sizes, lower recall and higher precision in Merge$_{Approx}$-Rank$_{Ionic}$ compared to Merge$_{Ionic}$-Rank$_{Ionic}$. The Doric unit rankings shows Doric spreads its faulty unit all over the ranked lists as it fails to consider collaborations. This explains why Doric might reach to its first faulty unit quickly, but fails to locate subsequent faulty units at the same rate compared to Hera techniques.

```
83.     int len = Character.codePointCount(input, 0, input.length());
84.      while (pos < len) {
             if {
                 ...
89.          }
90.          else {
                 ...
101.         }
102.     pos++;
103.     }
```

Figure 4.15: Source code drawn from Lang-17 from Defects4J.

Cluster	IonicCL	Cluster Rank	Unit rank for 1st faulty unit	Unit rank for 2nd faulty unit	Unit rank for 3rd faulty unit
75, 76, 79, 82, 83, 84, 85, 86, 87, 88, 89, 102, 103, 104	2.34^{-12}	4	22	23	24

Figure 4.16: Ranked list of clusters for Lang-17 using technique Merge$_{Ionic}$-Rank$_{Ionic}$

Cluster	IonicCL	Cluster Rank	Unit rank for 1st faulty unit	Unit rank for 2nd faulty unit	Unit rank for 3rd faulty unit
89, 104	2.34^{-12}	7	16	-	-
75, 76, 79, 82, 83, 84, 85, 86, 87, 88, 102, 103	1.7^{-12}	11	-	31	32

Figure 4.17: Ranked list of clusters for Lang-17 using technique Merge$_{Approx}$-Rank$_{Ionic}$

Unit	DoricCL	Unit Rank
89	2.34^{-12}	16
83	1.7^{-12}	26
102	1.7^{-12}	26

Figure 4.18: Ranked list of clusters for Lang-17 using technique Doric

4.8 Threats to validity

Defects4J might not be representative. Therefore, it may not be valid to generalize the claims made on these benchmarks to other programs.

Ionic considers collaboration of 'unit's. A unit could be any source code element such as a statement, a branch or a path. Our evaluation was limited to collaboration between lines of codes. Therefore, we cannot generalize these results for other granularity. However, using a higher granular unit like a path, may suffer from data sparsity as Holmes [4].

Defects4J benchmarks do not provide ground truth on whether there are collaborating faulty units in in a version. Therefore, we assumed every version with multiple faulty units is a version with multiple collaborating faulty units. Our last research question was whether Hera is better at localizing collaborating faulty units than existing FLTs. The answer for

this question was based on the aforementioned subset of versions. Thus, there is threat to the validity of the answer of this research question.

There may be defects in the implementation of this work. But we minimized this threat by validating our results with results from other FLTs.

Chapter 5

Related Work

There are many relevant work available to our work. But we only present a discussion about some of the most relevant work.

Context aware statistical debugging [7] creates a list of connected clusters of multiple units. Their main goal is to answer *how* to find the bug in addition to the commonly answered question *where* to find the bug. Each cluster comprises predicates that behave similarly from a statistical perspective. Later on, clusters are connected using a greedy approach to connect as many buggy predicate clusters as possible forming paths. Since the final output of this tool has clusters of connected units, this work also facilitates blame for multiple units to some extent. Note, their clusters simply have similar units and not the cohesive units that collaborated together to cause the failure. Connecting clusters on a greedy approach may lose their collective meaning further. Whereas our approach, Hera incorporates the collaborative causal likelihood creating clusters itself and guide the whole process based on collaborations.

Holmes [4] also explored incorporating context sensitivity (branches, paths) in FLTs. Its *unit* is a path. Hence, it collects path profiles. These path profiles are scored based on their suspiciousness. Additionally, since path profiles already contain branch information as well, they produce a ranked list of branch suspiciousness too. Since branches and paths contain multiple lines, this can be considered an attempt to blame multiple lines together as the cause of the failure. The quality of the suspiciousness of a path is highly dependent on many factors such as path length and number of actual faulty lines in the path. It also suffers from data sparsity because when the granularity of the *unit* increases, the execution frequencies decreases. In contrast, Hera has the potential to identify a subset of faulty units from a test case trace without limiting itself to a specific granularity level such as a path, branch or a statement.

Another related work [11], which is a variant of SBFL, produces a ranked list of functions. But their *units* are not functions. Instead, they use intra-procedural line patterns and their execution frequencies to infer the suspiciousness of a function. They first identify the line patterns within a function from call traces. Later, these initial patterns are mined

to identify subsets of patterns which occur frequently. The initial patterns and subsequent patterns are combined to create a full set of of patterns for a function. Then, each pattern is given a suspiciousness score based on the execution frequencies. A function's suspiciousness score is the the maximum suspiciousness score of any of its patterns. This approach uses collaborative information as it identifies patterns of multiple lines from execution profiles. However, they rank *functions* based on that information. In contrast, Hera is an FLT which produces a ranked list of clusters of intra-procedural units.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

IonicCL, the Collaborative Causal Likelihood calculation formula we introduced, is capable of incorporating collaborative information of programs. Hera, the new FLT we introduced, localizes more faulty units when it incorporates the most amount of collaborative information (technique Merge_{Ionic}-Rank_{Ionic}). Further, Hera localizes defects with multiple faulty units better than conventional FLTs.

6.2 Future work

Hera's clustering and ranking is parameterizable. We only explored three merging policies and two ranking policies in this evaluation. As future work, other merging or ranking policy could be explored.

There are other ways of using Hera. We can combine Hera with other semantics of programs using machine learning. The potential is investigated [21]. They use a Support Vector Machine to combine results from different FLTs. The weights are applied to tune the belief of each FLT.

Some Hera techniques produce clusters containing high noise. Unit level ranking within a cluster can be applied to highlight the units with higher causal likelihood. This may assist a developer to focus on highly faulty units first.

Currently, the capabilities of automated bug fixing is limited to create patches for defects requiring a single line fix. This is because the input they get is limited to unit level ranked lists from existing FLTs. If we could input the Hera output to such automated program repairing tools it may enhance the quality of the process by enabling multi-line fixes.

Bibliography

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, Sep. 2007.

[2] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82:1780–1792, 2009.

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[4] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering*, pages 34–44, May 2009.

[5] Ryan Chmiel and Michael C. Loui. Debugging: From novice to expert. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 17–21, New York, NY, USA, 2004. ACM.

[6] E. T. Jaynes. *Probability theory: The logic of science*. Cambridge University Press, Cambridge, 2003.

[7] Lingxiao Jiang and Zhendong Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 184–193, New York, NY, USA, 2007. ACM.

[8] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477, May 2002.

[9] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[10] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the*

2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[11] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. Fine-tuning spectrum based fault localisation with frequent method item sets. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 274–285, New York, NY, USA, 2016. ACM.

[12] David Landsberg and Earl T. Barr. Doric: Foundations for statistical fault localisation. *CoRR*, abs/1810.00798, 2018.

[13] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[14] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, August 2011.

[15] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[16] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 609–620, Piscataway, NJ, USA, 2017. IEEE Press.

[17] F. Steimann and M. Frenkel. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 121–130, Nov 2012.

[18] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 314–324, New York, NY, USA, 2013. ACM.

[19] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[20] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, March 2014.

[21] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

Appendix A

Proofs of Ionic

Doric framework introduced DoricCL, a formula to calculate causal likelihood of a unit causing an error (2.3). We extend Doric framework to Ionic and then introduce IonicCL. IonicCL calculates causal likelihood of *multiple units collaborating* to cause an error (IonicCL 3.1). The core of the proofs is from Doric on [12]. We extend these definitions and proofs where necessary to prove Ionic.

A.1 Preliminary definitions

Unit : A unit is a concrete element in source code. Many levels of units have been explored in related work such as statements, branches, basic blocks, paths, functions, and classes [1, 9, 4, 11]. U is the set of units. In Figure A.1, we considered $U = \{u_2, u_4, u_6, u_8, e\}$. e is the error(failure), which a test case covers if it fails. $U* = U - e$ and $U_{|U|} = e$

Test Suite(T) : A collection of test cases is a test suite T. A test case has a set of inputs to the program and an oracle which determines whether the output returned by the test case is expected or not. If the output is expected, then it is considered as passed, otherwise failed. In Figure A.1, $t_1, t_2, t_3, t_4, t_5, t_6$, are the test cases in test suite T.

Coverage Matrix: Suppose a the test suite (T) and a set of units (U). A coverage matrix (c) is a Boolean matrix of height $|T|$ and width $|U|$, where for each $u_i \in U$ and $t_k \in T$,

Definition A.1.1. $c_{i,k} = \begin{cases} 1, & \text{if } t_k \text{ covers } u_i, \\ 0, & \text{otherwise} \end{cases}$

Collaborating Causal Units : suppose a failing program has a set of units U. If there is a $\tilde{U}$ $\subseteq$ U where all $u_i \in \tilde{U}$,

1. Must be executed to cause the failure
2. Must be fixed to fix the bug

then , $\tilde{U}$ is a set of *Collaborating Causal Units* of the program. In Figure A.1, consider units 4 and 8 are such Collaborating Causal Units.

Collaborating Causality : Causality of a failure caused by collaborating units.

```
0     public int Bar(p, q){
1     if ( q > 500 && q < 1000 )
2         q = q-500;
3     if ( p > 500 && p < 1000 )
4         p = p-600;  //BUG
5     if ( q > 0 && q < 100 )
6         r = p/q ;
7     if ( q < 1000 )
8         s = q/p;    //BUG
9     ...
10    }
```

(a) Bar.java

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

(b) Coverage matrix

Figure A.1: Bar.java and its coverage matrix

A.2 Causal Models

Causal models are a set of models that captures the entire universe of possibilities that could have caused the failures. Suppose $j \in \mathbb{N}$ and m_j a matrix. Then $m^j_{i,k}$ is the value of the cell located at the ith column and kth row of matrix m^j , where this value is in $\{1, 0, \bullet\}$. Rows represent test cases and the columns represent units. $m^j_{i,k} = 0$ denotes u_i was not executed, hence cannot be a cause of the error e. 1 denotes u_i was executed but was not a cause of e. $\bullet$ denotes u_i was executed by t_k and was a cause of e. Suppose a the test suite (T), set of units (U), test case result (e) and Coverage matrix (c). Causal models defined as below.

Consider the coverage matrix in A.2

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Figure A.2: Coverage matrix

When T is a test suite and U a set of units, the set of models for a coverage matrix c is a set of matrices $M = m^1, ..., m^{|M|}$ of height $|T|$ and width $|U|$ satisfying:

Definition A.2.1. . $m^j_{i,k} \in \begin{cases} \{1, \bullet\} & \text{if } c_{i,k}e_k = 1 \text{ and } i \neq |U|, \\ c_{i,k} & \text{otherwise} \end{cases}$

where for each $t_k \in T$ there is some $u_i \in U$ such that $m^j_{i,k} = \bullet$. Intuitively, each causal model represents one possible way of causing the failures of test cases.

Number of causal models possible $= \prod_{k \in \tilde{T}} 2^{\rho_k} - 1,$ where $\tilde{T} \subseteq T$ and $\forall t_k \in \tilde{T}, e_k = 1$. Number of models for motivating example $= (2^4 - 1) * (2^3 - 1) = 105$. Some of those causal models are shown in Figure A.3.

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
\bullet & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
\bullet & \bullet & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
\bullet & 1 & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
\bullet & \bullet & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
1 & \bullet & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
1 & \bullet & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
\bullet & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
\bullet & \bullet & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
\bullet & 1 & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
\bullet & \bullet & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
1 & \bullet & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

$$
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
1 & \bullet & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6
\end{array}
\begin{array}{c}
\begin{array}{ccccc} u_2 & u_4 & u_6 & u_8 & e \end{array} \\
\begin{bmatrix}
\bullet & \bullet & 1 & 1 & 1 \\
1 & 1 & 0 & \bullet & 1 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad \cdots\cdots
$$

Figure A.3: Example of causal models for coverage matrix for Bar.java A.2

A.3 Syntax and semantics

Since the causal models capture the universe of possibilities how failures could have occurred, a probability calculation can be framed as model counting problem as below. Therefore, Doric introduced a language (L) to frame the hypotheses, such as,“Which unit was faulty” or “Which caused the error in which test case?” Suppose we have a hypothesis. Then they

show that a hypothesis can be framed as a model counting problem using L. The these counts are used to calculate probability of a hypothesis.

Definition A.3.1. When u_i has the reading "the ith unit was execute" *basic propositions* U is defined as $U = \{u_1, u_2, .., u_{|U|}\}$

Definition A.3.2. When h_i has the reading "the ith unit was *a* cause of the error" *partial causal hypotheses* H is defined as $H = \{h_1, h_2, .., h_{|U|}\}$

Definition A.3.3. When H_i has the reading "the ith unit was *the* cause of the erro" *total causal hypotheses* H_i is defined as $H_i = h_i \wedge \neg h_j$, where $h_j \in H - \{h_i\}$

Doric presented L, a language to frame the hypothesis. The hypotheses we would like test using L should be able to frame collaboration of units as well. One example would be, "Which units collaborated to cause the failure?" Therefore, L is extended with additional definitions.

Definition A.3.4. We define *Co-execution*, $u_{\tilde{U}}$, when U = {u_1, u_2,...,$u_{|U|}$}
$u_{\tilde{U}} = [\ \bigwedge\ u_i\ ]$, where $\tilde{U} \subseteq U$.
$u_{\tilde{U}}$ is read as the "units in $\tilde{U}$ were executed together". Each u_i in $\tilde{U}$ captures the fact that u_i was executed. The conjunction of all u_i in $\tilde{U}$ represents, "all the u_is in $\tilde{U}$ executed together".

Based on definition A.3.4 we define a new domain, U', to capture all possible ways units in U could co-execute.
$U' = \{\ u_{\tilde{U}}\ \}$, for each $\tilde{U} \subseteq U$.
$U' = \{[\bigwedge\ u_i\ ]\}$, for each $u_i \in \tilde{U}$ and for each $\tilde{U} \subseteq U$.
$U' = \{\ [u_1], [u_1 \wedge u_2], ..., [u_1 \wedge u_2 \wedge ... \wedge u_{|U|}], ..., [u_2], ..., [u_2 \wedge u_3 \wedge ... \wedge u_{|U|}], ..., [u_{|U|}]\}$.
$[u_i]$'s reading has the same reading as u_i in Doric, "ith unit was executed". $[u_1 \wedge u_2]$ is read as "*1*st and *2*nd units co-executed". $[u_1 \wedge u_2 \wedge ... \wedge u_{|U|}]$ is read as "Units 1 to $|U|$ co-executed".

Definition A.3.5. we define *Collaborative partial causal hypothesis*, $h_{\tilde{U}}$, when H = {h_1, h_2,...,$h_{|U|}$}
$h_{\tilde{U}} = [\ \bigwedge\ h_i\ ]$, where $\tilde{U} \subseteq U$. This corresponds to partial causal hypothesis set $\tilde{H} \in$ H for all $h_i \in \tilde{H}$. $h_{\tilde{U}}$ is read as the "Collaborating units in $\tilde{U}$ was *a* cause of the error". Each h_i in $\tilde{H}$ captures the hypothesis where unit i was a cause of the failure. The conjunction of all $h_i \in \tilde{H}$ represents a hypothesis "all of the h_is $\in \tilde{H}$ happened together, meaning collaboratively they are *a* cause of the failure".

Based on definition A.3.5 we define a new domain H' to capture all possible ways UUTs in U could have collaborated to cause the failure.
$H' = \{h_{\tilde{U}}\}$, for each $\tilde{U} \subseteq U$
$H' = \{[\bigwedge\ h_i\ ]\}$ for each $h_i \in \tilde{U}$ and for each $\tilde{U} \subseteq$ U.
$H' = \{\ [h_1], [h_1 \wedge h_2], ..., [h_1 \wedge h_2 \wedge ... \wedge h_{|U|}], ..., [h_2], ..., [h_2 \wedge h_3 \wedge ... \wedge h_{|U|}], ..., [h_{|U|}]\}$.
$[h_i]$'s reading has the same reading as h_i in Doric, "ith unit was *a* cause of the error". $[h_1 \wedge h_2]$ is read as "Collaborating *1*st and *2*nd units were *a* cause of the error". $[h_1 \wedge h_2 \wedge ... \wedge h_{|U|}]$ is read as "Collaborating units 1 to $|U|$ were *a* cause of the error".

Definition A.3.6. We define *Collaborative total causal hypothesis*, $H_{\tilde{U}}$
$H_{\tilde{U}} = h_{\tilde{U}} \wedge \neg\ h_j$, for all $h_j \in$ {H - $\tilde{H}$ }.

$H_{\tilde{U}}$ intuitively captures the hypothesis "Collaborating u_is in set $\tilde{U}$ were the *exclusive cause of the* error. Nor any other unit or a collaboration of units caused the error".

Definition A.3.7. Valuations
(1) $v_k(u_i) = \{m^j \in M | m^j_{i,k} \in \{1, \bullet\}\}$, for $u_i \in U$
(2) $v_k(h_i) = \{m^j \in M | m^j_{i,k} = \bullet\}$, for $h_i \in H$
(3) $v_k(\phi \wedge \psi) = v_k(\phi) \cap v_k(\psi)$, for $\phi, \psi \in L$
(4) $v_k(\neg\phi) = M - v_k(\phi)$, for $\phi \in L$
(5) $v_k(\diamond_n \phi) = v_n(\phi)$, for $\phi \in L$
(6) $v_k(u_{\tilde{U}}) = \{m^j \in M | m^j_{i,k} \in \{1, \bullet\}\}$, for $u_i \in \tilde{U}$
(7) $v_k(h_{\tilde{U}}) = \{m^j \in M | m^j_{i,k} = \bullet\}$, for $h_i \in \tilde{H}$
$v_k(\phi)$ is read "the models where ϕ in the kth test case". The definitions are explained with the example below.

Example A.3.1.
Suppose, U = {u_1, u_2, u_3 }. Then,
H = {h_1, h_2, h_3 }
U' = { [u_1], [$u_1 \wedge u_2$], [$u_1 \wedge u_3$], [$u_1 \wedge u_2 \wedge u_3$], [$u_2$], [$u_2 \wedge u_3$], [$u_3$] }
H' = { [h_1], [$h_1 \wedge h_2$], [$h_1 \wedge h_3$], [$h_1 \wedge h_2 \wedge h_3$], [$h_2$], [$h_2 \wedge h_3$], [$h_3$] }
Suppose we select $\tilde{U}$ = {u_1, u_2}. Then,
$\tilde{H}$ = {h_1, h_2}. Then,
$u_{\tilde{U}} = u_{\tilde{u_1, u_2}} = [\bigwedge u_{\tilde{u_1, u_2}}] = [u_1 \wedge u_2]$
$h_{\tilde{U}} = h_{\tilde{h_1, h_2}} = [\bigwedge h_{\tilde{h_1, h_2}}] = [h_1 \wedge h_2]$
H - $\tilde{H}$ = {h_1, h_2, h_3 } - {h_1, h_2} = { h_3 }
$H_{\tilde{U}} = h_{\tilde{U}} \wedge \neg h_j$, for all $h_j \in$ {H - $\tilde{H}$ }, by definition A.3.6.
$H_{\tilde{U}} = [h_1 \wedge h_2] \wedge \neg h_3$

A.4 Probability theory

To determine the probability of a hypothesis, Doric presents theory of probability.

Definition A.4.1. Let M, L, V be non-empty set of models, a language and set of valuations respectively. Then a *probability theory* is a tuple <**P**,P,w>, where

1. w : $2^M \to \mathbb{R}$
2. $\mathbf{P} = \{P_1, ..., P_|T|\}$ is a set of *probability functions*, where for each $v_k \in V$ there is some $P_k \in \mathbf{P}$ such that
   $P_k(\phi) = \frac{w(v_k(\phi))}{w(M)}$
3. P is the *expected likelihood* function defined as follows.
   $P(\phi) = \sum_k \frac{P_k(\phi)}{|T|}$, for each $\phi \in L$

$P_k(\phi)$ is the "probability that ϕ occurred in the kth test case" and $P(\phi)$ is "the expected likelihood that ϕ".

Definition A.4.2. Let f be a function, which intuitively measures the frequency in which a proposition is true, in a test suite. Doric defines,
$f(\phi) = \sum_k(\phi)$, where $f_k(\phi) = 1$ if $v_k(\phi) = \text{M}$ and 0 otherwise.
Then, For all $\phi \in L^*, P(\phi) = f(\phi)/|T|$.

$P(f_i)$, the "likelihood of the ith unit was the fault (fault likelihood)" can be calculated as, $P(f_i) = P(\forall_{k=1}^{|T|} \diamond_k h_i)$
Intuitively, this is the proportion of models where the u_i caused the error. i.e the proportion of models a $\bullet$ in the ith column.

In probability theory, if there are n mutually exclusive possibilities, and there is no sufficient reason to believe one over the other, then their relative likelihoods are equal. This is referred to as *principle of indifference*. The probability of one event happening becomes $1/n$ [6]. When ρ_t is the number of units executed in a failing test case k,

$$P_t(h_{\tilde{U}}) = \begin{cases} 1/(2^{\rho_t} - 1), & \text{if } c_{i,t}e_t = 1, \\ 0, & \text{otherwise} \end{cases}$$

A.5 IonicCL

The unit which gives $\max_{u_i \in U^*} P(f_i)$ will be the most likely to cause an error. But, if that unit has low propensity to cause the error, then that unit should not be the first unit to be examined. Instead, the unit with highest propensity should be given priority. To capture the concept of propensity, Doric introduces *Causal Likelihood* (CL), "the likelihood a given unit was the cause of an error when it was executed, $P(H_i|u_i)$".

It can be calculated by generating all the causal models, and counting them to come up with the proportions. But this is intractable. Doric introduced the likelihood equation 2.3 . When c is a coverage matrix, and $\rho_k = \Sigma_{u,j \in U*} c_{j,k}$ (which is the number of units executed by t_k), then for each $u_i \in U*$

$$P(H_i|u_i) = \frac{\sum_k \{c_{i,k}e_k/(2^{\rho_k} - 1)\}}{\sum_k c_{i,k}} \tag{A.1}$$

We prove the DoricCL equation A.1 can be extended to IonicCL (A.2). New equation provides collaborating causal likelihood. $P(H_{\tilde{U}}|u_{\tilde{U}})$ is read as "the likelihood of the set of collaborating units in $\tilde{U}$, were the exclusive cause of the error, given that they were executed together".

When $\tilde{U} \in$ U,

$$IonicCL = P(H_{\tilde{U}}|u_{\tilde{U}}) = \frac{\sum_k \{\prod_{i \in \tilde{U}} c_{i,k}e_k/(2^{\rho_k} - 1)\}}{\sum_k \prod_{i \in \tilde{U}} c_{i,k}e_k} \tag{A.2}$$

A.6 IonicCL proofs

Definition A.6.1. Suppose for each $m^j \in M$ and $t_k \in T, M_k = \{m^j_k | 1 \leqslant j \leqslant |M|\}$. M_k refers to all **different** kth rows of all models.
Concatenation of two matrices is denoted by $M_i \cdot M_j = \{x \cdot y | x \in M_i \wedge y \in M_j\}$.
Therefore, $M = M_1 \cdot M_2 \cdot ...M_{|T|}$.
$|M_k| = 2^{\rho_k} - 1$ if t_k fails, 1 otherwise

When $\tilde{U} \subseteq$ U,

Proof of A.2
Followed by definition of conditional probability

$$P(H_{\tilde{U}} | u_{\tilde{U}}) = \frac{P(H_{\tilde{U}} \wedge u_{\tilde{U}})}{P(u_{\tilde{U}})} \tag{A.3}$$

$P(H_{\tilde{U}} | u_{\tilde{U}}) = \frac{P(H_{\tilde{U}})}{P(u_{\tilde{U}})}$, by definitions A.3 and A.4
$P(H_{\tilde{U}}) = \sum_k \frac{P_k(H_{\tilde{U}})}{|T|}$, by definition A.5
$P(u_{\tilde{U}}) = \sum_k \frac{\prod_{i \in \tilde{U}} c_{i,k}}{|T|}$, by definition A.8
Then, $P(H_{\tilde{U}} | u_{\tilde{U}}) = \frac{\sum_k P_k(H_{\tilde{U}})}{\sum_k \prod_{i \in \tilde{U}} c_{i,k}}$ by cancellation.
$P_k(H_{\tilde{U}}) = \prod_{i \in \tilde{U}} c_{i,k} e_k / (2^{\rho_k} - 1)$, by definition A.7.
Therefore, $P(H_{\tilde{U}} | u_{\tilde{U}}) = \frac{\sum_k \prod_{i \in \tilde{U}} c_{i,k} e_k / (2^{\rho_k} - 1)}{\sum_k \prod_{i \in \tilde{U}} c_{i,k}}$.

$P(H_{\tilde{U}} \wedge u_{\tilde{U}})$ is "probability of a set of collaborating units were the cause of the error and they were executed together".

$$P(H_{\tilde{U}} \wedge u_{\tilde{U}}) = P(H_{\tilde{U}}) \tag{A.4}$$

Proof A.4:
Based on valuations A.3.7, it is sufficient to show $v_k(H_{\tilde{U}} \wedge u_{\tilde{U}}) = v_k(H_{\tilde{U}})$
Thus, it is sufficient to show $v_k(H_{\tilde{U}} \wedge u_{\tilde{U}}) = v_k(H_{\tilde{U}}) \cap v_k(u_{\tilde{U}})$, by definition A.3.7
Now, $v_k(H_{\tilde{U}}) \cap v_k(u_{\tilde{U}}) \subseteq v_k(H_{\tilde{U}})$ as in general $X \cap Y \subseteq X$
Thus it remains to show $v_k(H_{\tilde{U}}) \cap v_k(u_{\tilde{U}}) \subseteq v_k(H_{\tilde{U}})$
It is sufficient to show $v_k(u_{\tilde{U}}) \subseteq v_k(H_{\tilde{U}})$ as in general $X \subseteq X$
Now Suppose $\tilde{U} = \{$u$_1$, u$_2\}$ without losing generality
$H_{\tilde{U}} = [h_1 \wedge h_2] \wedge \neg\ h_j$, for all $h_j \in \{$H - $\tilde{H}$ $\}$, by definition A.3.6
Then, $v_k(H_{\tilde{U}}) = v_k(h_1 \wedge h_2 \wedge \neg h_3 ... \wedge \neg h_{|U|-1})$,by definition A.3.7
$v_k(H_{\tilde{U}})) = v_k(h_1) \cap v_k(h_2) \cap v_k(\neg h_3) \cap ...v_k(\neg h_{|U|-1}))$
$v_k(H_{\tilde{U}})) = \{\{m^j | m^j_{1,k} = \bullet\} \cap \{m^j | m^j_{1,k} \in \{1,0\}\} \cap \{m^j | m^j_{2,k} = \bullet\} \cap \{m^j | m^j_{2,k} \in \{1,0\}\} \cap ...\}$
$v_k(H_{\tilde{U}})) = \{\{m^j | m^j_{1,k} = \bullet \cap m^j_{1,k} \in \{1,0\}\} \cap \{m^j | m^j_{2,k} = \bullet \cap m^j_{2,k} \in \{1,0\}\} \cap ...\}$
$v_k(H_{\tilde{U}})) = \{\{m^j | m^j_{1,k} \in \{1,\bullet\} \cap m^j_{1,k} = \bullet m^j_{1,k} \in \{1,0\} \{m^j | m^j_{2,k} \in \{1,\bullet\} \cap m^j_{2,k} = \bullet \cap m^j_{2,k} \in \{1,0\} ...\}$
$v_k(u_i) = \{m^j \in M | m^j_{i,k} \in \{1,\bullet\}\}$, for $u_i \in U$, by definition A.3.7

Hence, $v_k(u_1) \subseteq v_k(H_{\tilde{U}}))$ and $v_k(u_2) \subseteq v_k(H_{\tilde{U}}))$
Therefore, $v_k(u_1) \cap v_k(u_2) \subseteq v_k(H_{\tilde{U}}))$
Therefore, $P(H_{\tilde{U}} \wedge u_{\tilde{U}}) = P(H_{\tilde{U}})$

$P(H_{\tilde{U}})$ is "Expected likelihood of a set of collaborating UUTs were the cause of the error".

$$P(H_{\tilde{U}}) = \sum_k \frac{P_k(H_{\tilde{U}})}{|T|} \tag{A.5}$$

Proof of A.5 : This is followed by the expected likelihood function A.4.1.

$P_t(H_{\tilde{U}})$= "Probability of a set of collaborating units were the cause of the error for the given test case t".

$$P_t(H_{\tilde{U}}) = \begin{cases} 1/(2^{\rho_t} - 1), & \text{if } \prod_{i \in \tilde{U}} c_{i,t} e_t = 1, \\ 0, & \text{otherwise} \end{cases} \tag{A.6}$$

Proof of A.6.
We prove that "probability of a set of collaborating units were the cause of the error for the given test case t", can be calculated similarly to the calculation "probability of a u_i was the cause of the error for the given test case t" from Doric. We only need to consider all units in $\tilde{U}$ in the new calculation.
$P_t(H_i) = |v_t(H_i)|/|M|$ by definition A.4.1.
$P_t(H_{\tilde{U}}) = |v_t(H_{\tilde{U}})|/|\mathrm{M}|$ by definition A.4.1.
Let's prove the top condition, $P_t(H_{\tilde{U}}) = 1/(2^{\rho_t} - 1), \text{if } \prod_{i \in \tilde{U}} c_{i,t} e_t = 1$ first.
Without loss of generality let k=1 and $\tilde{U} = \{u_i,\, u_q\}$.
Assume $c_{i,1} = c_{q,1} = e_1 = 1$.
M = $M_1 \cdot M_2 \cdot ... \cdot M_{|T|}$ by A.6.1 and causal models remain constant for Doric and Ionic.
Thus, $v_1(H_{\tilde{U}}) = \{m_1^j | m^j \in v_1(H_{\tilde{U}})\} \cdot M_2 \cdot M_n$ as in general $v_1(\phi) = \{m_k^j \in M_1 | m^j \in v_1(H_{\tilde{U}}))\} \cdot M_2 \cdot ... \cdot M_n\}$.
So, $|v_1(H_{\tilde{U}})| = |\{m_1^j | m^j \in v_1(H_{\tilde{U}})\}| \times |M_2| \times ... \times |M_{|T|}|$
So, $|v_1(H_{\tilde{U}})| = 1 \times |M_2| \times ... \times |M_{|T|}|$, given if $m^j, m^k \in v_1(H_i)$ then $m_1^j = m_1^k$. Note that there is only one unique way to model a given set of units exclusively causing the error in a test case. This is independent of the hypothesis.
Thus, $(1 \times |M_2| \times ... \times |M_{|T|}|)/(|M_1| \times |M_2| \times ... \times |M_{|T|}|)$.
$(1 \times |M_2| \times ... \times |M_{|T|}|)/(|M_1| \times |M_2| \times ... \times |M_{|T|}|) = 1/|M_1|$, by cancellation.
$1/|M_1| = 1/(2^{\rho_t} - 1)$ given $\rho_1 = \sum_{i<|U|} c_{i,1}$.
Now, let's consider the bottom condition, $P_t(H_{\tilde{U}}) = 0$ when either $c_{i,1}$ or $c_{q,1}$ or $e_1 = 0$.
Accordingly, there are no models in $v_1(h_i)$ or $v_1(h_q)$, by definition A.3.7.
Thus, no models in $v_1(H_{\tilde{U}})$ by definition A.3.7 and $H_{\tilde{U}}$.
So, $P_1(H_{\tilde{U}}) = 0$

$$P_k(H_{\tilde{U}}) = \prod_{i \in \tilde{U}} c_{i,k} e_k / (2^{\rho_k} - 1) \tag{A.7}$$

Proof of A.7.
Assume, $\prod_{i \in \tilde{U}} c_{i,k} e_k = 1$.
$P_k(H_{\tilde{U}}) = 1/(2^{\rho_k} - 1)$, by first condition of equation A.6
$1/(2^{\rho_k} - 1) = \prod_{i \in \tilde{U}} c_{i,k} e_k/(2^{\rho_k} - 1)$ by our assumption.
Assume, $\prod_{i \in \tilde{U}} c_{i,k} e_k \neq 1$.
Then $P_k(H_{\tilde{U}}) = 0$ by second condition of equation A.6
In order for this to be true either any of c_i, k or e_k should be 0.
Therefore, $\prod_{i \in \tilde{U}} c_{i,k} e_k/(2^{\rho_k} - 1) = 0$.
Thus, $P_k(H_{\tilde{U}}) = \prod_{i \in \tilde{U}} c_{i,k} e_k/(2^{\rho_k} - 1)$

$P(u_{\tilde{U}})$= Expected likelihood of UUTs in $\tilde{U}$ getting executed together.

$$P(u_{\tilde{U}}) = \sum_k \frac{\prod_{i \in \tilde{U}} c_{i,k}}{|T|} \tag{A.8}$$

Proof of A.8.
$P(u_{\tilde{U}}) = f(u_{\tilde{U}})/|T|$, by definition A.4.2 and probability theory is generic enough as mentioned in sec A.4
$P(u_{\tilde{U}}) = \sum_k f_k(u_{\tilde{U}})/|T|$ by definition A.4.1 and probability theory is generic enough as mentioned in sec A.4
Given c is a Boolean matrix, it remains to show $f_k(u_{\tilde{U}}) = \prod_{i \in \tilde{U}} c_{i,k}$ for both cases when $\prod_{i \in \tilde{U}} c_{i,k} = 1$ or 0.
Assume $\prod_{i \in \tilde{U}} c_{i,k} = 1$.
Then for all $m \in M$ and $i \in \tilde{U}$, $m_{i,k} \in \{1, \bullet\}$, by definition A.2.1.
Thus, $v_k(u_{\tilde{U}}) = M$, by definition A.3.7.
So, $f_k(u_{\tilde{U}}) = 1$.
Assume $\prod_{i \in \tilde{U}} c_{i,k} = 0$.
Then for all $m \in M, m_{i,k} = 0$ by definition A.2.1.
Thus, $v_k(u_{\tilde{U}}) = M$, by definition A.3.7.