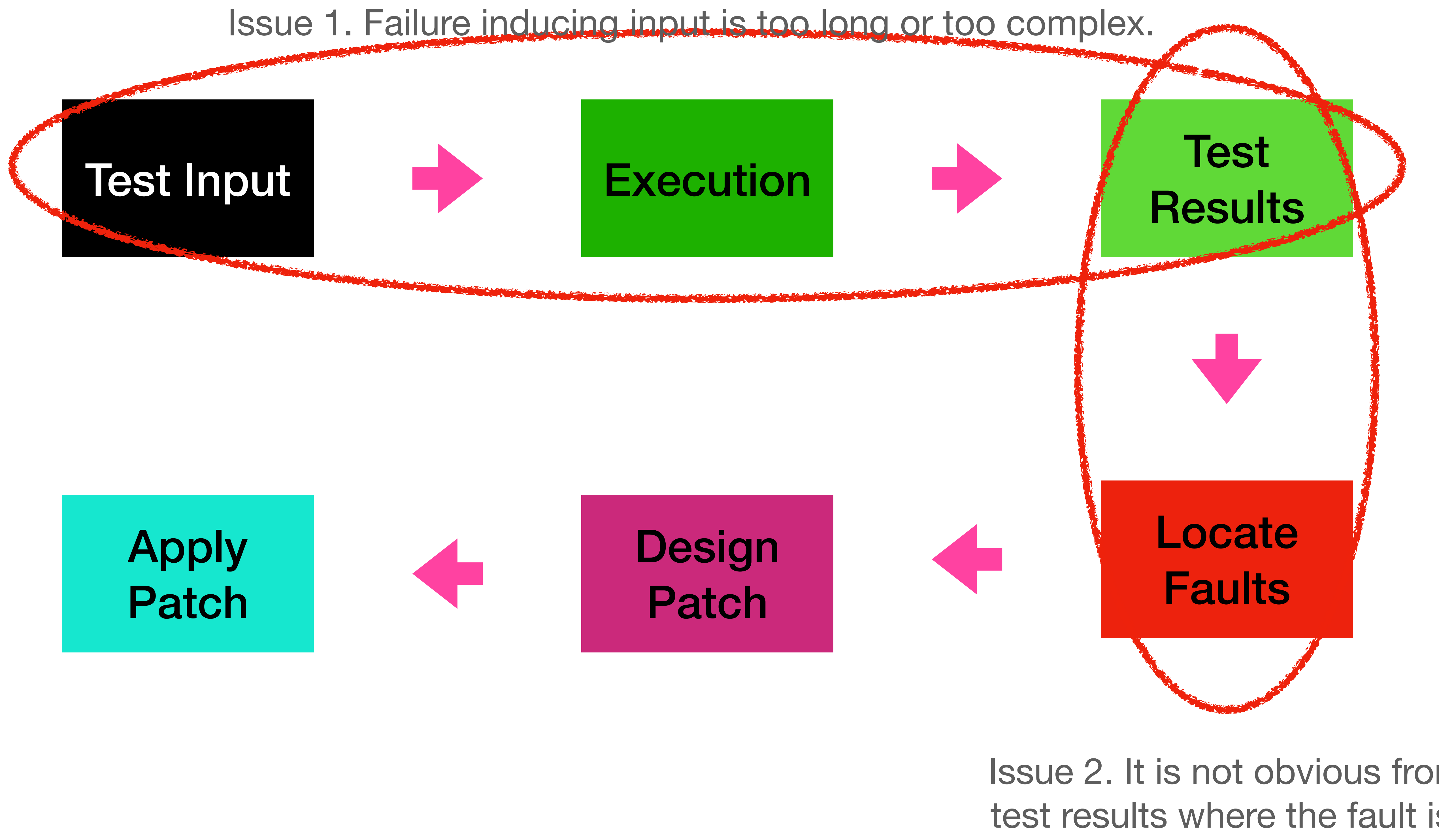
Fault Localisation

CS453 Automated Software Engineering

Shin Yoo | COINSE@KAIST

Process of Debugging


Issue 1. Failure inducing input is too long or too complex.
Test Input
Execution
Test Results
Locate Faults
Design Patch
Apply Patch
Issue 2. It is not obvious from test results where the fault is.

Fault Localisation

- Hard for humans: we increasingly have to work on and with large code base written by others.
- Hard for machines: automated repair techniques rely heavily on automated fault localisation.

Fault Localisation

- Machines are bad at semantic reasoning (yet).
- We need to convert the fault localisation problem into something quantitative, something less than the full semantic.
- How do we reason about the location of a fault, using available information?
    - What are the available information?

Fault Localisation

- Here, we consider four different techniques.
  - Delta debugging (a slightly different approach, but highly relevant)
  - Information Retrieval Technique
  - Spectrum Based Fault Localisation
  - Mutation Based Fault Localisation
- For a complete overview:
  - W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. IEEE Transactions on Software Engineering, 42(8):707, August 2016.

A Motivating Example

```
<td align=left valign=top>
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION
VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION
VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION
VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT></td> <td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

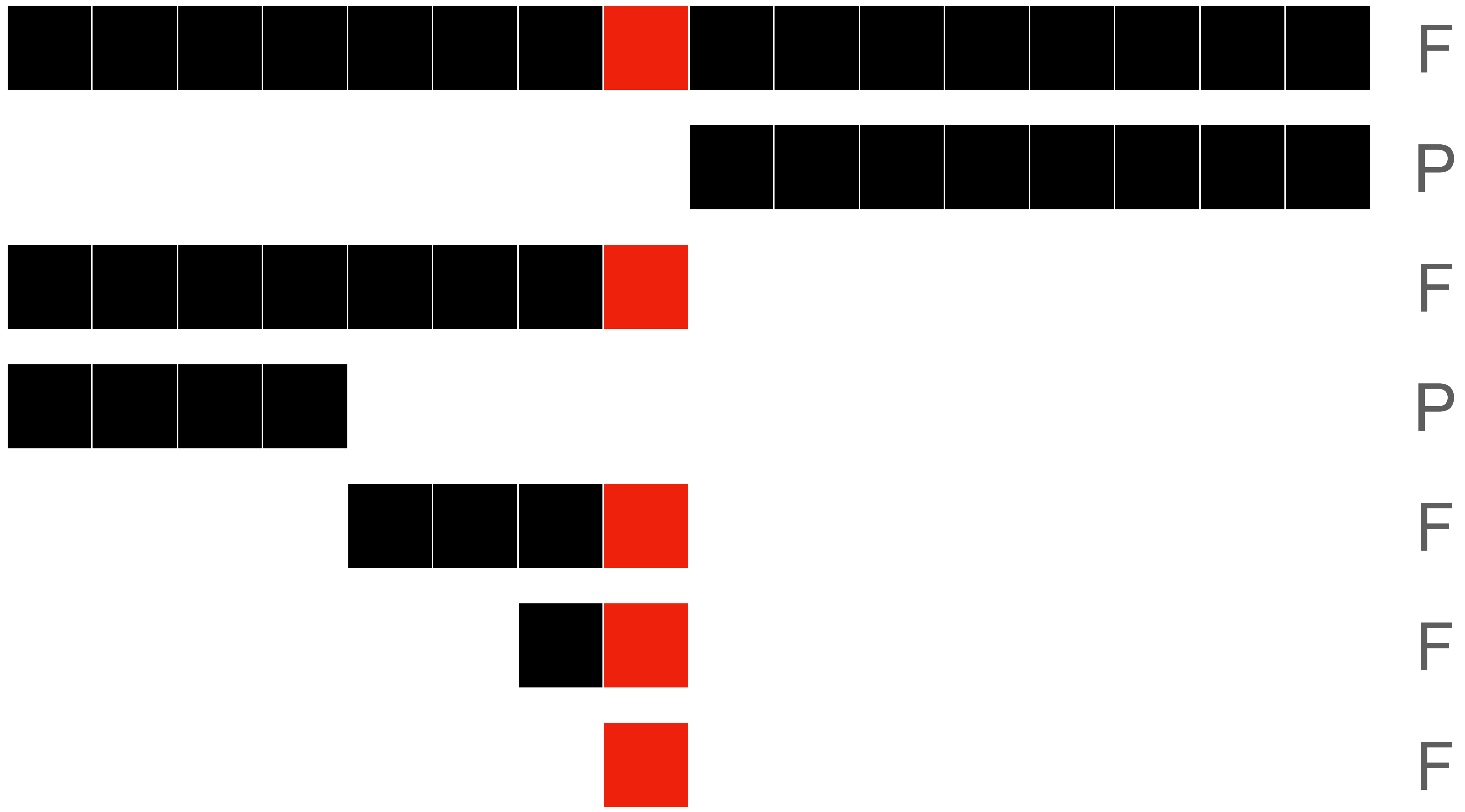
This HTML code caused Mozilla to crash: what is the actual cause?

Delta Debugging

- Delta debugging is a technique designed to minimise failure inducing inputs.
    - Given a long input that causes a program to fail, delta debugging returns the shortest input that still causes the failure.
    - It is based on a simple idea: binary search.

A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 28(2):183–200, Feb. 2002.

Delta Debugging



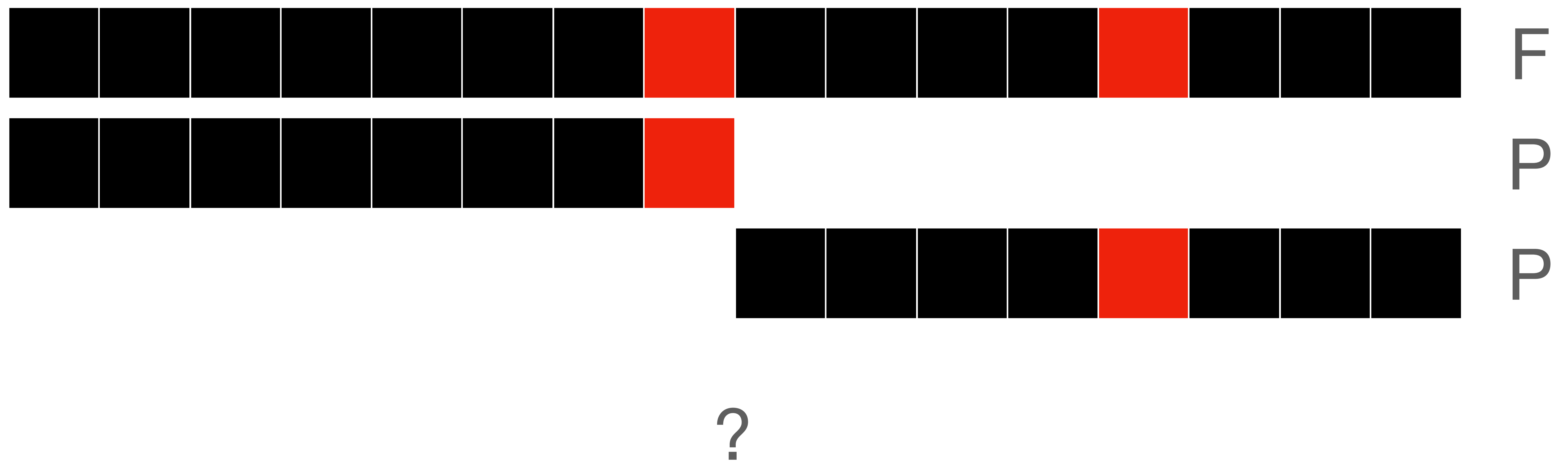
Delta Debugging

- The basic binary search version is simple:
    - Cut and throw away half of the input, and see if the output is still wrong
    - If so, go back to the first step
    - If not, go back to the previous state, and throw away the other half

Issues to Consider

- Given N parts, delta debugging requires O(logN) steps to find the root cause (binary search). Depending on problems, this may or may not be acceptable.
- It requires fully automated oracle.
- What if the root cause of the failure is actually composed of two separate parts?
- What if the failure inducing input cannot be cut in half?

DD: What if the root cause of the failure is actually composed of two separate parts?

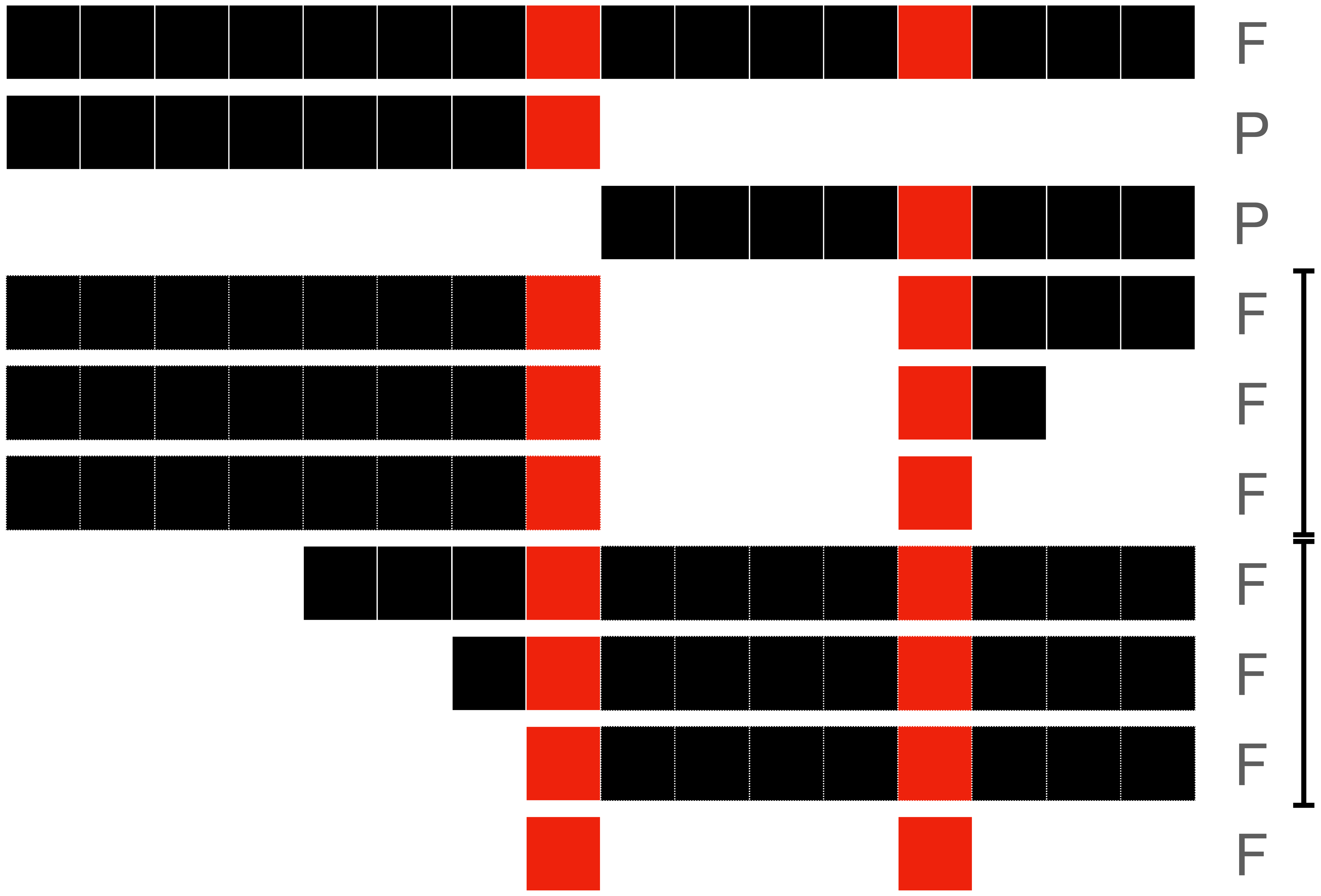


This is a not uncommon case. To deal with such disjoint causes,
the actual Delta Debugging algorithm is more than the basic binary search.

Recursive Delta Debugging

- Line 1: if there is a single input left, return it.
- Line 2-7: see if program still fails with halves of the given input - if it does, continue halving recursively
- Line 8-10: otherwise, make two recursive calls
    - First: keep the first half of the given input, and apply DD to the second half
    - Second: keep the second half of the given input, and apply DD to the first half

```
DD$(P, \{i_1, \ldots, i_n\})$
(1)     if $n == 1$ then return $i_1$
(2)     $P_1 = \left(P + \{i_1, \ldots, i_{\frac{n}{2}}\}\right)$
(3)     $P_2 = \left(P + \{i_{\frac{n}{2}} + 1, \ldots, i_n\}\right)$
(4)     if $P_1$ fails
(5)         return DD $\left(P, \{i_1, \ldots, i_{\frac{n}{2}}\}\right)$
(6)     else if $P_2$ fails
(7)         return DD $\left(P, \{i_{\frac{n}{2}} + 1, \ldots, i_n\}\right)$
(8)     else
(9)         return DD $\left(P_2, \{i_1, \ldots, i_{\frac{n}{2}}\}\right) +$
(10)        DD $\left(P_1, \{i_{\frac{n}{2}+1}, \ldots, i_n\}\right)$
```


F
P
P
F
F
F
F
F
F
F

DD: What if the failure inducing input cannot be cut in half?

- Suppose we are testing a C compiler. The following code actually made GCC 2.95.2 crash.
- What if we try to cut this faulty source code in two halves?
- Neither is a valid C code anymore, not to mention a failure inducing input.

```c
double mult( double z[], int n ) {
    int i;
    int j;
    for (j= 0; j< n; j++) {
        i= i+j+1;
        z[i]=z[i]*(z[0]+0);
    }
    return z[n];
}

int copy(double to[], double from[], int count)
{
    int n= (count+7)/8;
    switch (count%8) do {
        case 0: *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;

        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
    } while (--n > 0);
    return (int)mult(to,2);
}

int main( int argc, char *argv[] ) {
    double x[20], y[20];
    double *px= x;

    while (px < x + 20)
        *px++ = (px-x)*(20+1.0);
    return copy(y,x,20);
}
```

Hierarchical Delta Debugging

- When input is highly structured, simply cutting it in half is likely to invalidate the input structure, damaging the efficiency of DD.
    - Cutting C source code by halves will eventually work: DD will recurse down to the single character level, and find the structural boundaries that will work.
    - But this will take a significant amount of test execution that is just invalid.

G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In Proceedings of the 28th International Conference on Software Engineering, ICSE ’06, pages 142–151, New York, NY, USA, 2006. ACM.

Hierarchical Delta Debugging

- Hierarchical Delta Debugging (HDD) does not cut at the raw data level, but handles structural elements.
  - If the input is source code, HDD deletes Abstract Syntax Tree nodes, instead of half of the lines.
  - If the input is HTML, HDD deletes XML nodes.
- Instead of recursive binary search based deletions, HDD recursively goes into deeper nested structures.
  - For example, HDD first tries to delete an entire function from source code; if this fails, HDD then tries to delete half of the basic blocks inside the function, etc.
- HDD always produce syntactically correct output, and preserves atomic elements such as identifiers.

Delta Debugging (ddmin)

```
t(double z[], int)
{
  int i;
  int j;
  for(;;j++) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

HDD

```
mult(double *z, int n)
{
  int i;
  int j;
  for (;;) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

DD is sensitive to whitespaces and deletion boundaries.

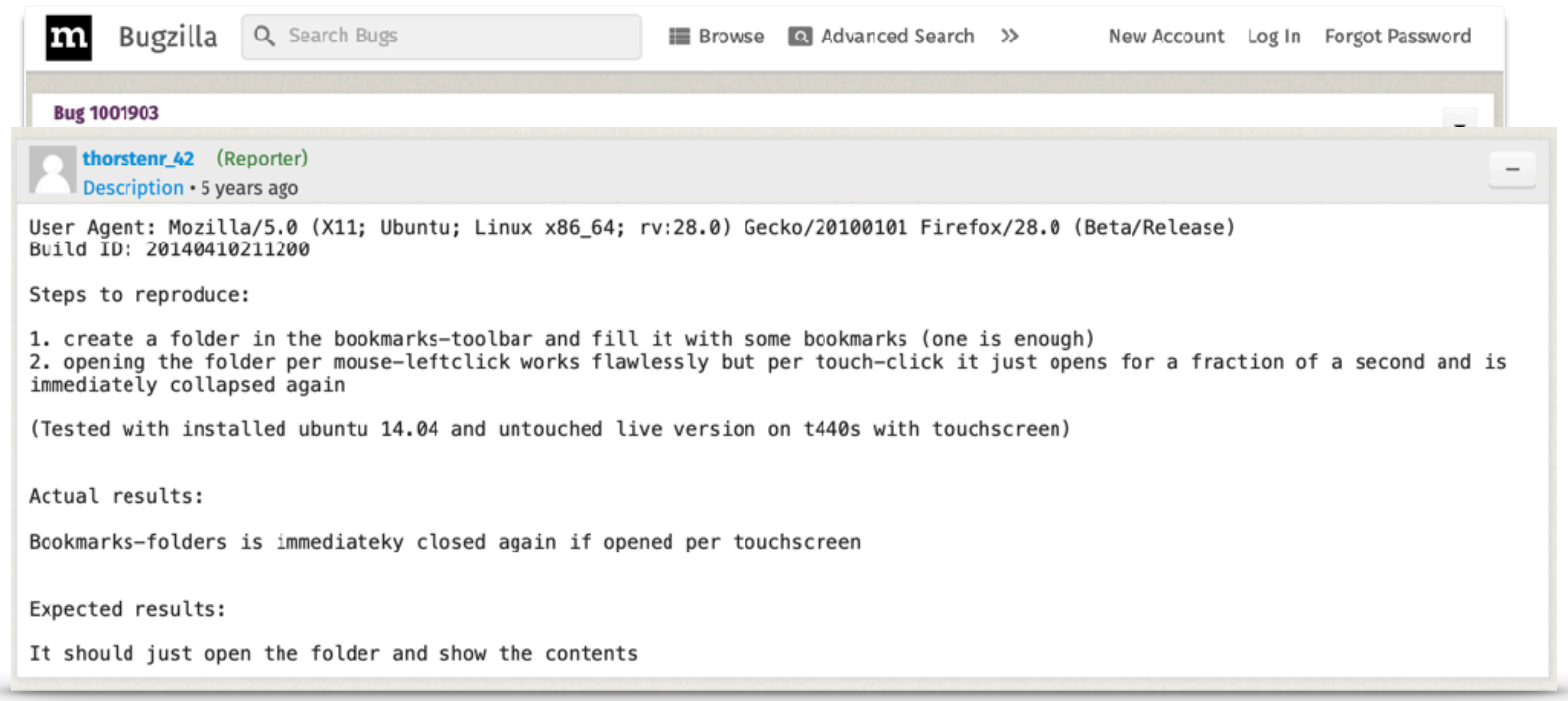
Consider the `int`, or `j++`: in both cases, ddmin can delete them only when its binary search deletion has the deletion size of 3 and matches these characters exactly.

Note that ddmin also reduced `mult` to `t`.

Delta Debugging Tools

- DD.py (https://www.st.cs.uni-saarland.de/dd/DD.py) and tutorial (https://www.st.cs.uni-saarland.de/dd/ddusage.php3)
- delta (http://delta.tigris.org) which implements HDD
- Lithium (http://www.squarefree.com/lithium/using.html), a python implementation of DD
- Git Bisect (https://git-scm.com/docs/git-bisect), which performs DD to a series of git commits

Let’s have a look at a real bug report



```
User Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:28.0) Gecko/20100101 Firefox/28.0 (Beta/Release)
Build ID: 20140410211200

Steps to reproduce:

1. create a folder in the bookmarks-toolbar and fill it with some bookmarks (one is enough)
2. opening the folder per mouse-leftclick works flawlessly but per touch-click it just opens for a fraction of a second and is
immediately collapsed again

(Tested with installed ubuntu 14.04 and untouched live version on t440s with touchscreen)


Actual results:

Bookmarks-folders is immediateky closed again if opened per touchscreen


Expected results:

It should just open the folder and show the contents
```

https://bugzilla.mozilla.org/show_bug.cgi?id=1001903

FireFox Bug 1001903

- After creating a folder in the bookmark toolbar and a bookmark inside the folder, mouse left-click successfully opens the folder but touch-click fails to open it: it opens but immediately closes itself.
- While we do not know the internals of FireFox, it is easy to guess which parts of the overall system is most likely to contain this fault.

FireFox Bug 1001903

- After creating a folder in the bookmark toolbar and a bookmark inside the folder, mouse left-click successfully opens the folder but touch-click fails to open it: it opens but immediately closes itself.
- While we do not know the internals of FireFox, it is easy to guess which parts of the overall system is most likely to contain this fault.

IR based Fault Localisation

- Information Retrieval techniques help a user to quickly obtain resources relevant to an information need, from a large collection of information resources.
- Fault localisation can be thought of as finding a resource (a program element) that is relevant to an information need (the reported symptoms of the failure), from a large collection of information resources (the entire system).
  - The bug report becomes our query.
  - The entire source code becomes our collection of documents.
  - Fault localisation is to find the source code that matches the bug report the best.

How do we do the matching?

- There are many ways to represent documents in IR: here, we are going to look at one of the basic model, called Vector Space Model (VSM).
- VSM allows us to represent queries and documents as vectors.
- This, in turn, allows us to calculate distances between our query and documents easily.

Vector Space Model

- First, define the vocabulary, a set of words that are meaningful to us.
- Given a vocabulary with *N* terms, we represent both the query and the documents as vectors:

$$d_j = (w_{1,j}, w_{2,j}, \ldots, w_{N,j})$$

$$q = (w_{1,q}, w_{2,q}, \ldots, w_{N,q})$$

- The dimensionality of these vectors is equal to the number of terms in the vocabulary, *N*.
- If the term w_2 appears in d_j, $w_{2,j}$ is a non-zero number; otherwise, it is a zero. There are many ways to set the non-zero number: we are going to study *tf-idf*.

Tf-Idf

- Term frequency of a term *t* in document *d*, *tf(t, d)*, is simply the number of times *t* appears in *d*.
- Inverse document frequency of a term *t* in a set of documents *D*, *idf(t, D)*, is the logarithmically scaled inverse fraction of the documents that contain the term *t*.
    - A higher *idf(t, D)* means that only a few documents in *D* contains the term *t*: in other words, *t* is not common.
    - A lower *idf(t, D)* means that most of the documents in *D* contain the term *t*: in other words, *t* is very common.

$$idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

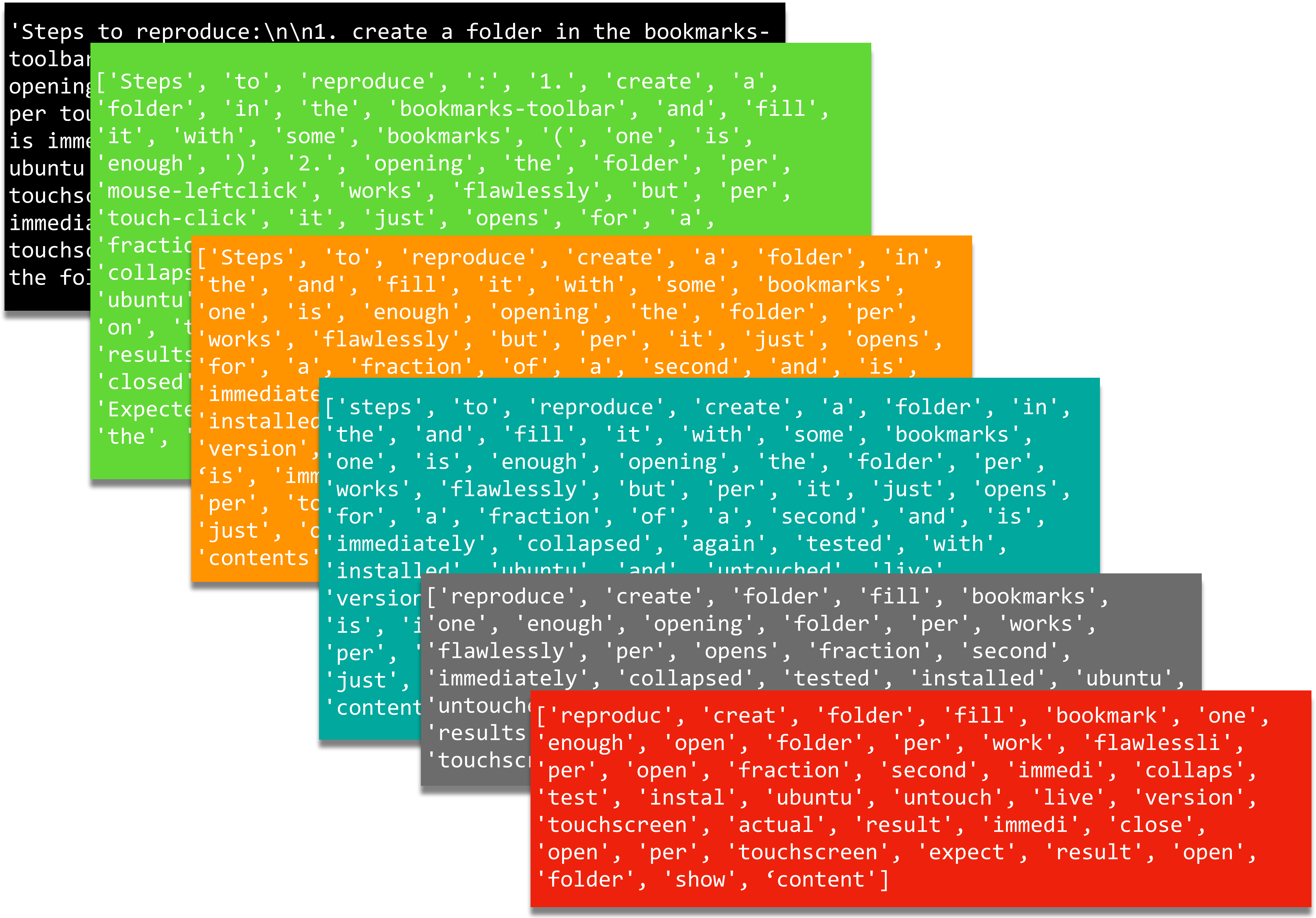
Tf-Idf

- Tf-Idf is simply the product of *tf(t, d)* and *idf(t, D)*.
    - A high *tfidf(t, d, D)* means that *t* happens specifically in *d*, but not commonly in *D*. In other words, *t* is unique to *d*, and not other documents in *D*.
    - A low *tfidf(t, d, D)* means that either *t* does not appear frequently in *d*, or if it does then it appears frequently in most of the documents in *D*. In other words, *t* is not unique to *d*.

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D)$$

Preparing bug reports for IR based Fault Localisation

- We follow the standard text cleansing process used in many IR and machine learning applications.
    - Tokenisation: we break down the bug report to a list of word tokens.
    - Remove punctuation: for example, “file’s” becomes “file”
    - Case normalisation: for example, “File” and “FILE” all become “file”
    - Stop word filtering: some words are extremely common in English (e.g., “a”, “an”, “and”…) and best left out. These words are called stop words.
    - Stemming: reduce each word to its root (e.g., “opens” and “opening” both become “open”).
- There are many widely used libraries that will perform all of these steps.



```
'Steps to reproduce:\n\n1. create a folder in the bookmarks-
toolbar
opening
per tou
is imme
ubuntu
touchsc
immedia
touchsc
the fol
```

```
['Steps', 'to', 'reproduce', ':', '1.', 'create', 'a',
'folder', 'in', 'the', 'bookmarks-toolbar', 'and', 'fill',
'it', 'with', 'some', 'bookmarks', '(', 'one', 'is',
'enough', ')', '2.', 'opening', 'the', 'folder', 'per',
'mouse-leftclick', 'works', 'flawlessly', 'but', 'per',
'touch-click', 'it', 'just', 'opens', 'for', 'a',
'fractio
'collaps
'ubuntu'
'on', 't
'results
'closed'
'Expecte
'the', '
```

```
['Steps', 'to', 'reproduce', 'create', 'a', 'folder', 'in',
'the', 'and', 'fill', 'it', 'with', 'some', 'bookmarks',
'one', 'is', 'enough', 'opening', 'the', 'folder', 'per',
'works', 'flawlessly', 'but', 'per', 'it', 'just', 'opens',
'for', 'a', 'fraction', 'of', 'a', 'second', 'and', 'is',
'immediate
'installed
'version',
'is', 'imm
'per', 'to
'just', 'c
'contents'
```

```
['steps', 'to', 'reproduce', 'create', 'a', 'folder', 'in',
'the', 'and', 'fill', 'it', 'with', 'some', 'bookmarks',
'one', 'is', 'enough', 'opening', 'the', 'folder', 'per',
'works', 'flawlessly', 'but', 'per', 'it', 'just', 'opens',
'for', 'a', 'fraction', 'of', 'a', 'second', 'and', 'is',
'immediately', 'collapsed', 'again', 'tested', 'with',
'installed', 'ubuntu', 'and', 'untouched', 'live',
'versio
'is', 'i
'per', '
'just',
'content
```

```
['reproduce', 'create', 'folder', 'fill', 'bookmarks',
'one', 'enough', 'opening', 'folder', 'per', 'works',
'flawlessly', 'per', 'opens', 'fraction', 'second',
'immediately', 'collapsed', 'tested', 'installed', 'ubuntu',
'untouch
'results
'touchsc
```

```
['reproduc', 'creat', 'folder', 'fill', 'bookmark', 'one',
'enough', 'open', 'folder', 'per', 'work', 'flawlessli',
'per', 'open', 'fraction', 'second', 'immedi', 'collaps',
'test', 'instal', 'ubuntu', 'untouch', 'live', 'version',
'touchscreen', 'actual', 'result', 'immedi', 'close',
'open', 'per', 'touchscreen', 'expect', 'result', 'open',
'folder', 'show', 'content']
```

Stemming Applied

Preparing code for IR based Fault Localisation

- The standard text cleansing process also allies to source code.
- With source code, stop words should also include the reserved keywords of the used programming language.
- In addition, we usually apply additional normalisation to identifiers:
  - Camel case breakdown: for example, divide “openDialog” to “open” and “dialog”.
  - Snake case breakdown: for example, divide “create_bookmarks” to “create” and “bookmarks”

VSM and Distance

- After cleansing, using a predefined vocabulary, we can calculate *tfidf(t, d, D)* for each *t* (word) and *d* (the query or a source code file). This converts the query and each source code file to a vector.

- Given two vector, one representing the query and another representing a source code file, the distance between two vectors can be calculated using the cosine distance:

$$cos(d_j, q) = \frac{d_j \cdot q}{\|d_j\|\|q\|} = \frac{\sum_{i=1}^{N} w_{i,j} w_{i,q}}{\sqrt{\sum_{i=1}^{N} w_{i,j}^2}\sqrt{\sum_{i=1}^{N} w_{i,q}^2}}$$

- Choose the file that has the shortest distance to the query, and it is more likely to contain the fault that is responsible for the symptom described in the query (i.e., the bug report).

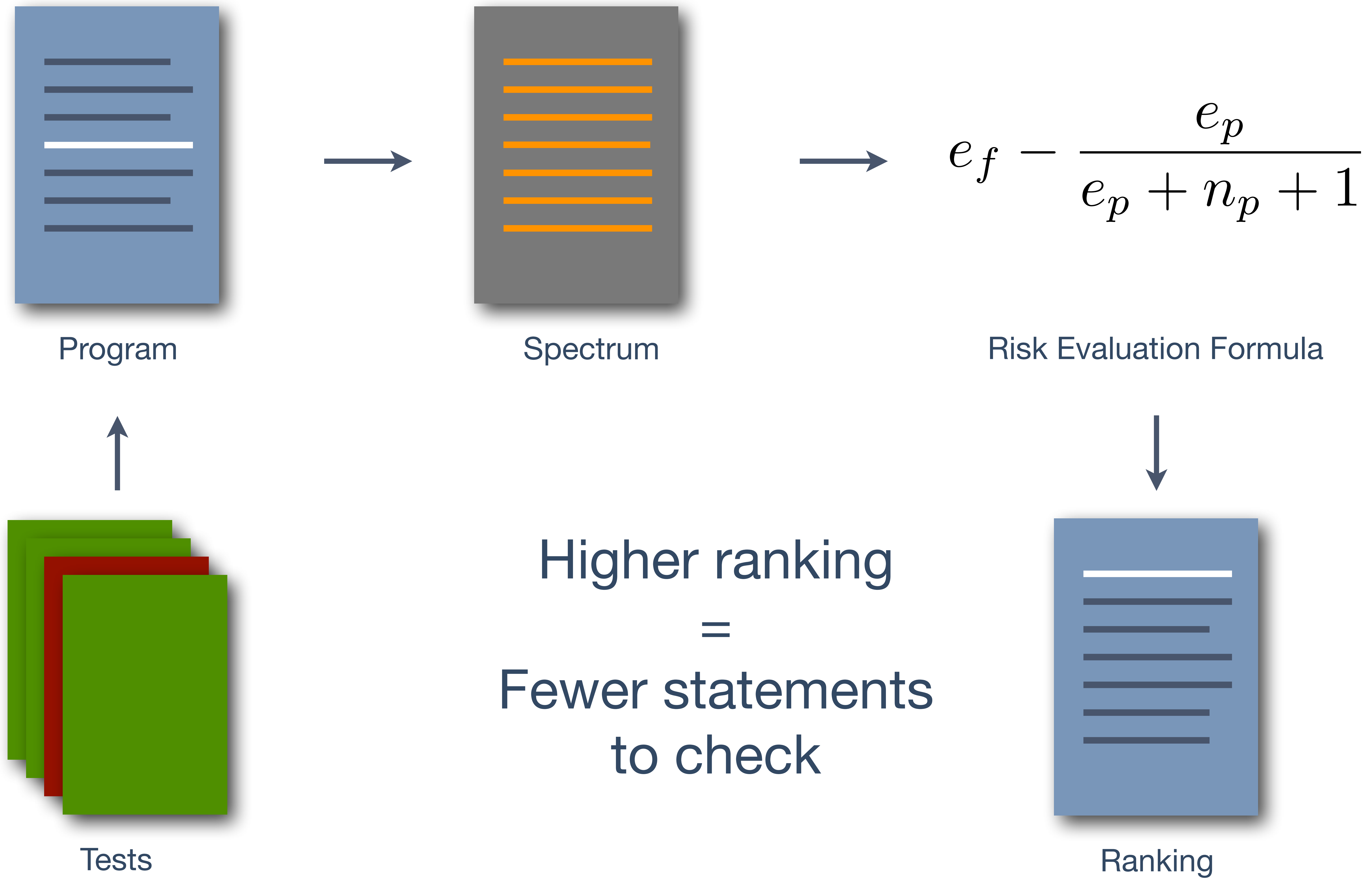
IR Based Fault Localisation

- Strengths
    - Simple and available: many existing work on IR already improved the accuracy of textual queries - we get free ride!
    - Intuitive, easy to understand.
- Weaknesses
    - Requires a (very good and detailed) bug report.
    - There is inherent limit to accuracy, because the document to be matched (i.e., the unit of localisation) needs to be of certain size (otherwise lexical similarity becomes more random).

Spectrum Based Fault Localisation

- Intuition: If there is a faulty line, executing it is likely to increase the probability of a failure. Similarly, not executing the faulty line is likely to decrease the probability of a failure.
- This decision process
  - requires only test results and structural coverage, but
  - is essentially a statistical inference, and does not give any guarantee
- Question: how do we formalise the above intuition in a computable form?

Spectrum Based Fault Localisation



$$e_f - \frac{e_p}{e_p + n_p + 1}$$

Higher ranking
=
Fewer statements
to check

Spectrum Based Fault Localisation

- Program Spectrum: for each structural unit (i.e. statements or branches), summarise the test result AND coverage into a tuple of the following four numbers
  - ep: # of test cases that execute this unit and pass
  - ef: # of test cases that execute this unit and fail
  - np: # of test cases that do not execute this unit and pass
  - nf: # of test cases that do not execute this unit and fail

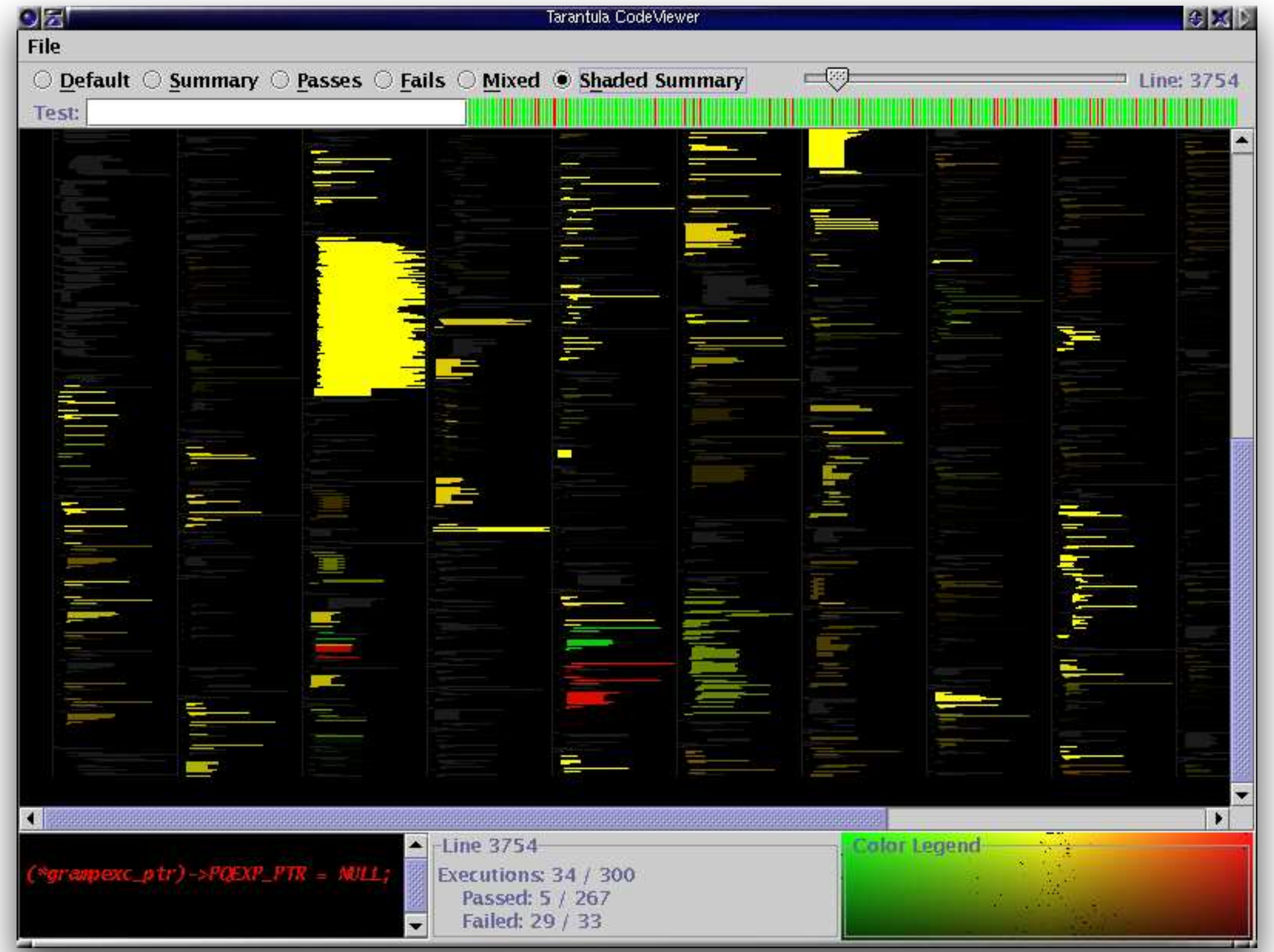
Spectrum Based Fault Localisation

Structural Elements	Test t_1	Test t_2	Test t_3	Spectrum e_p	e_f	n_p	n_f	Tarantula	Rank
s_1	●			1	0	0	2	0.00	9
s_2	●			1	0	0	2	0.00	9
s_3	●			1	0	0	2	0.00	9
s_4	●			1	0	0	2	0.00	9
s_5	●			1	0	0	2	0.00	9
s_6	●		●	1	1	0	1	0.33	4
s_7 (faulty)		●	●	0	2	1	0	1.00	$\bigcirc$ 1
s_8	●	●		1	1	0	1	0.33	4
s_9	●	●	●	1	2	0	0	0.50	2
Result	P	F	F						

$$\text{Tarantula} = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p} + \frac{e_f}{e_f+n_f}}$$

Tarantula (Jones & Harrold, 2001)

- Originally meant as a visualization technique, later the poster child of SBFL.



```
(*grampexc_ptr)->PQEXP_PTR = NULL;
```

Line 3754
Executions: 34 / 300
Passed: 5 / 267
Failed: 29 / 33

State of the Art c.a. 2010

$$\frac{2e_f}{e_f + n_f + e_p}$$

$$\frac{e_f}{e_f + n_p + 2(e_p + n_f)}$$

$$\frac{e_f}{e_f + n_f + e_p}$$

$$\frac{2(e_f + n_p)}{2(e_f + n_p) + e_p + n_f}$$

$$\frac{e_f}{e_f + 2(n_f + e_p)}$$

$$\frac{e_f}{n_f + e_p}$$

Over 30 formulæ in the literature: none guaranteed to perform best for all types of faults

$$\frac{e_f + n_p}{n_f + e_p}$$

$$\frac{e_f}{e_f + n_f + e_p + n_p}$$

$$\frac{e_f + n_p}{e_f + n_f + e_p + n_p}$$

$$\frac{2e_f}{2e_f + n_f + e_p}$$

$$\frac{1}{2}(\frac{e_f}{e_f + n_f} + \frac{e_f}{e_f + e_p})$$

$$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p} + \frac{e_f}{e_f+n_f}}$$

$$\frac{e_f + n_p - n_f - e_p}{e_f + n_f + e_p + n_p}$$

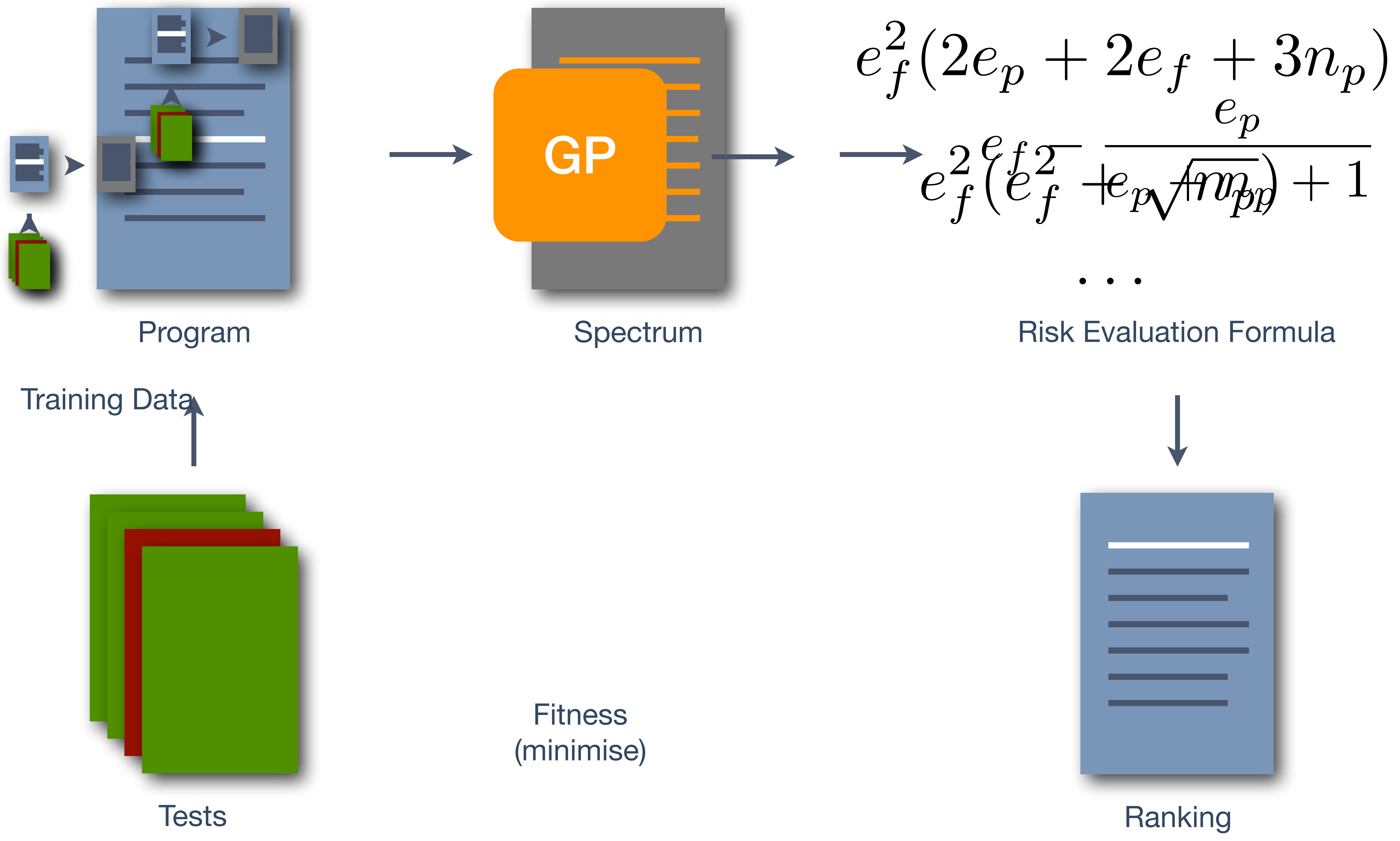
Name	Formula	Name	Formula
ER1$_a$	$\begin{cases}-1 & \text{if } n_f>0\\ n_p & \text{otherwise}\end{cases}$	ER1$_b$	$e_f-\frac{e_p}{e_p+n_p+1}$
ER5$_a$	$e_f-\frac{e_f}{e_p+n_p+1}$	ER5$_b$	$\frac{e_f}{e_f+n_f+e_p+n_p}$
ER5$_c$	$\begin{cases}0 & \text{if } e_f<F\\ 1 & \text{otherwise}\end{cases}$	GP$_2$	$2(e_f+\sqrt{e_p+n_p})+\sqrt{e_p}$
Ochiai	$\frac{e_f}{(e_f+n_f)(e_f+e_p)}$	GP$_3$	$\sqrt{\lvert e_f^2-\sqrt{e_p}\rvert}$
Jaccard	$\frac{e_f}{e_f+n_f+e_p}$	GP$_{13}$	$e_f(1+\frac{1}{2e_p+e_f})$
AMPLE	$\lvert\frac{e_f}{F}-\frac{e_p}{P}\rvert$	GP$_{19}$	$e_f\sqrt{\lvert e_p-e_f+F-P\rvert}$
Hamann	$\frac{e_f+n_p-e_p-n_f}{P+F}$	Tarantula	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$
Dice	$\frac{2e_f}{e_f+e_p+n_f}$	RusselRao	$\frac{e_f}{e_p+e_f+n_p+n_f}$
M1	$\frac{e_f+n_p}{n_f+e_p}$	SørensenDice	$\frac{2e_f}{2e_f+e_p+n_f}$
M2	$\frac{e_f}{e_f+n_p+2n_f+2e_p}$	Kulczynski1	$\frac{e_f}{n_f+e_p}$
Hamming	e_f+n_p	Kulczynski2	$\frac{1}{2}(\frac{e_f}{e_f+n_f}+\frac{e_f}{e_f+e_p})$
Goodman	$\frac{2e_f-n_f-e_p}{2e_f+n_f+e_p}$	SimpleMatching	$\frac{e_f+n_p}{e_p+e_f+n_p+n_f}$
Euclid	$\sqrt{e_f+n_p}$	RogersTanimoto	$\frac{e_f+n_p}{e_f+n_p+2n_f+2e_p}$
Wong1	e_f	Sokal	$\frac{2e_f+2n_p}{2e_f+2n_p+n_f+e_p}$
Wong2	e_f-e_p	Anderberg	$\frac{e_f}{e_f+2e_p+2n_f}$

Wong3	$e_f-h, h=\begin{cases}e_p & \text{if } e_p\leq 2\\ 2+0.1(e_p-2) & \text{if } 2<e_p\leq 10\\ 2.8+0.01(e_p-10) & \text{if } e_p>10\end{cases}$
Ochiai2	$\frac{e_f n_p}{\sqrt{(ef+e_p)(n_f+n_p)(e_f+n_p)(e_p+n_f)}}$
Zoltar	$\frac{e_f}{e_f+e_p+n_f+\frac{10000n_f e_p}{e_f}}$

Parnin & Orso, ISSTA 2011

- Title: “Are Automated Debugging Techniques Actually Helping Programmers?”
- Based on human studies, authors say “no, not really”!
  - Some of the widely accepted assumptions, such as perfect bug understanding, may not hold in reality.
  - **BIGGEST IMPACT**: “percentage will not cut it” - later studies move away from the Expense metric.

Yoo, 2012: Evolving Formulas



$$e_f^2(2e_p + 2e_f + 3n_p)$$

$$e_f^2(e_f^2 + \sqrt{n_p})$$

$$\frac{e_f}{2} - \frac{e_p}{e_p + n_p} + 1$$

$$\ldots$$

The Competition

- We choose 9 formulæ from Naish et al. 2011:
  - Op2 is the known best.
  - Jaccard, Tarantula, Ochiai are widely studied in SE.
  - Wong & AMPLE are recent additions.

$$Op1 = \begin{cases} -1 & \text{if } n_f > 0 \\ n_p & \text{otherwise} \end{cases} \qquad Op2 = e_f - \frac{e_p}{e_p + n_p + 1}$$

$$Jaccard = \frac{e_f}{e_f + n_f + e_p}$$

$$Tarantula = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p} + \frac{e_f}{e_f+n_f}}$$

$$AMPLE = |\frac{e_f}{e_f + n_f} - \frac{e_p}{e_p + n_p}|$$

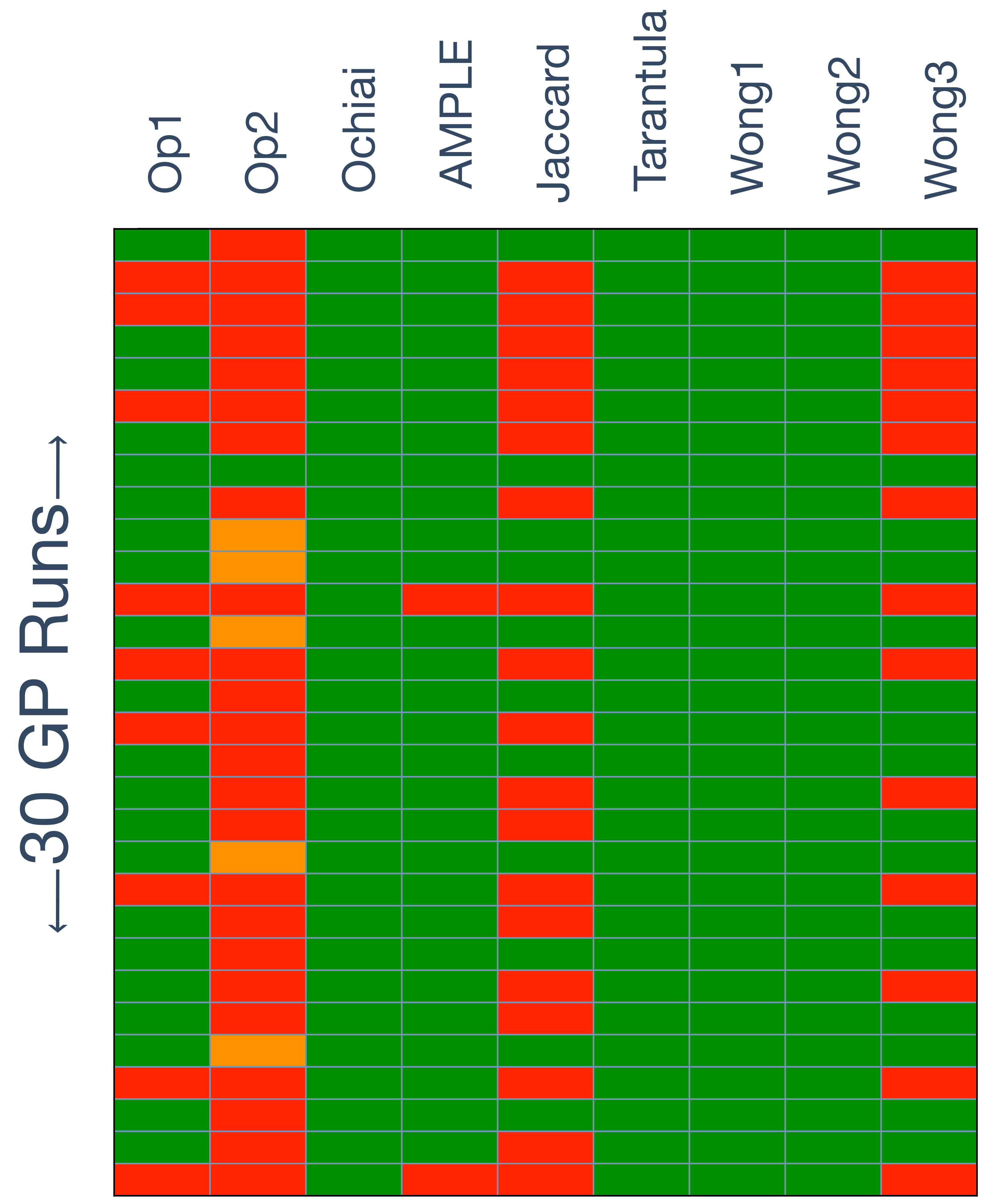
$$Ochiai = \frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}}$$

$$Wong1 = e_f \quad Wong2 = e_f - e_p$$

$$Wong3 = e_f - h, h = \begin{cases} e_p & \text{if } e_p \le 2 \\ 2 + 0.1(e_p - 2) & \text{if } 2 < e_p \le 10 \\ 2.8 + 0.001(e_p - 10) & \text{if } e_p > 10 \end{cases}$$

Results

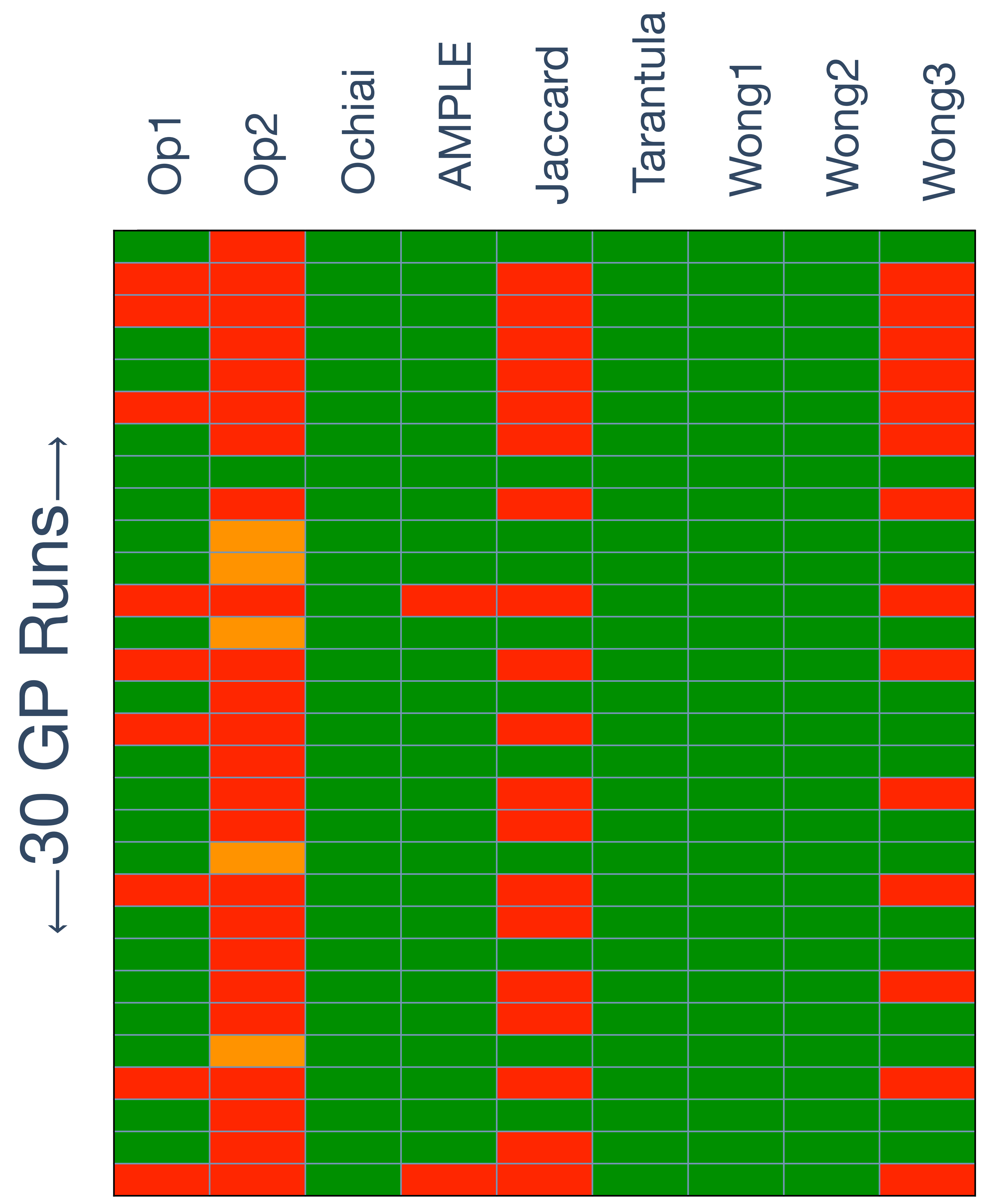
- Green: GP outperforms the other.
- Orange: GP exactly matches the other.
- Red: The other outperforms GP.



4 Unix tools w/ 92 faults: 20 for training, 72 for evaluation.

Results

- GP completely outperforms Ochiai, Tarantula, Wong 1 & 2, and mostly outperforms AMPLE.
- Op1, Jaccard, and Wong 3 are tough to beat.
- Op2 is very good but it is not impossible to do better.



4 Unix tools w/ 92 faults: 20 for training, 72 for evaluation.

Evolved Formulæ

ID	Refined Formula	ID	Refined Formula
GP01	$e_f(n_p + e_f(1 + \sqrt{e_f}))$	GP16	$\sqrt{e_f^{\frac{3}{2}} + n_p}$
GP02	$2(e_f + \sqrt{n_p}) + \sqrt{e_p}$	GP17	$\frac{2e_f + n_f}{e_f - n_p} + \frac{n_p}{\sqrt{e_f}} - e_f - e_f^2$
GP03	$\sqrt{\|e_f^2 - \sqrt{e_p}\|}$	GP18	$e_f^3 + 2n_p$
GP04	$\sqrt{\|\frac{n_p}{e_p - n_p} - e_f\|}$	GP19	$e_f\sqrt{\|e_p - e_f + n_f - n_p\|}$
GP05	$\frac{(e_f + n_p)\sqrt{e_f}}{(e_f + e_p)(n_p n_f + \sqrt{e_p})(e_p + n_p)\sqrt{\|e_p - n_p\|}}$	**GP20**	$2(e_f + \frac{n_p}{e_p + n_p})$
GP06	$e_f n_p$	GP21	$\sqrt{e_f + \sqrt{e_f + n_p}}$
GP07	$2e_f(1 + e_f + \frac{1}{2n_p}) + (1 + \sqrt{2})\sqrt{n_p}$	GP22	$e_f^2 + e_f + \sqrt{n_p}$
GP08	$e_f^2(2e_p + 2e_f + 3n_p)$	GP23	$\sqrt{e_f}(e_f^2 + \frac{n_p}{e_f} + \sqrt{n_p} + n_f + n_p)$
GP09	$\frac{e_f\sqrt{n_p}}{n_p + n_p} + n_p + e_f + e_f^3$	GP24	$e_f + \sqrt{n_p}$
GP10	$\sqrt{\|e_f - \frac{1}{n_p}\|}$	GP25	$e_f^2 + \sqrt{n_p} + \frac{\sqrt{e_f}}{\sqrt{\|e_p - n_p\|}} + \frac{n_p}{(e_f - n_p)}$
GP11	$e_f^2(e_f^2 + \sqrt{n_p})$	**GP26**	$2e_f^2 + \sqrt{n_p}$
GP12	$\sqrt{e_p + e_f + n_p - \sqrt{e_p}}$	GP27	$\frac{n_p\sqrt{(n_p n_f - e_f)}}{e_f + n_p n_f}$
GP13	$e_f(1 + \frac{1}{2e_p + e_f})$	GP28	$e_f(e_f + \sqrt{n_p} + 1)$
GP14	$e_f + \sqrt{n_p}$	GP29	$e_f(2e_f^2 + e_f + n_p) + \frac{(e_f - n_p)\sqrt{n_p e_f}}{e_p - n_p}$
GP15	$e_f + \sqrt{n_f + \sqrt{n_p}}$	GP30	$\sqrt{\|e_f - \frac{n_f - n_p}{e_f + n_f}\|}$

Theoretical Analysis of SBFL

- For all the gory details:
    - X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for Spectrum-Based Fault Localization. ACM Transactions on Software Engineering Methodology, 22(4):31:1– 31:40, October 2013.
    - X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman. Provably optimal and human-competitive results in SBSE for Spectrum Based Fault Localisation. In G. Ruhe and Y. Zhang, editors, Search Based Software Engineering, volume 8084 of Lecture Notes in Computer Science, pages 224–238. Springer Berlin Heidelberg, 2013.

Crash Course into Our Proof System

- Given a test suite and a formula R, a program PG = {s_1, s_2, ..., s_n} can be divided into the following three subsets:
  - S^R_B: set of elements that are ranked higher than the faulty element, s_f
  - S^R_F: set of elements that are tied to the faulty element, s_f
  - S^R_A: set of elements that are ranked lower than the faulty element, s_f

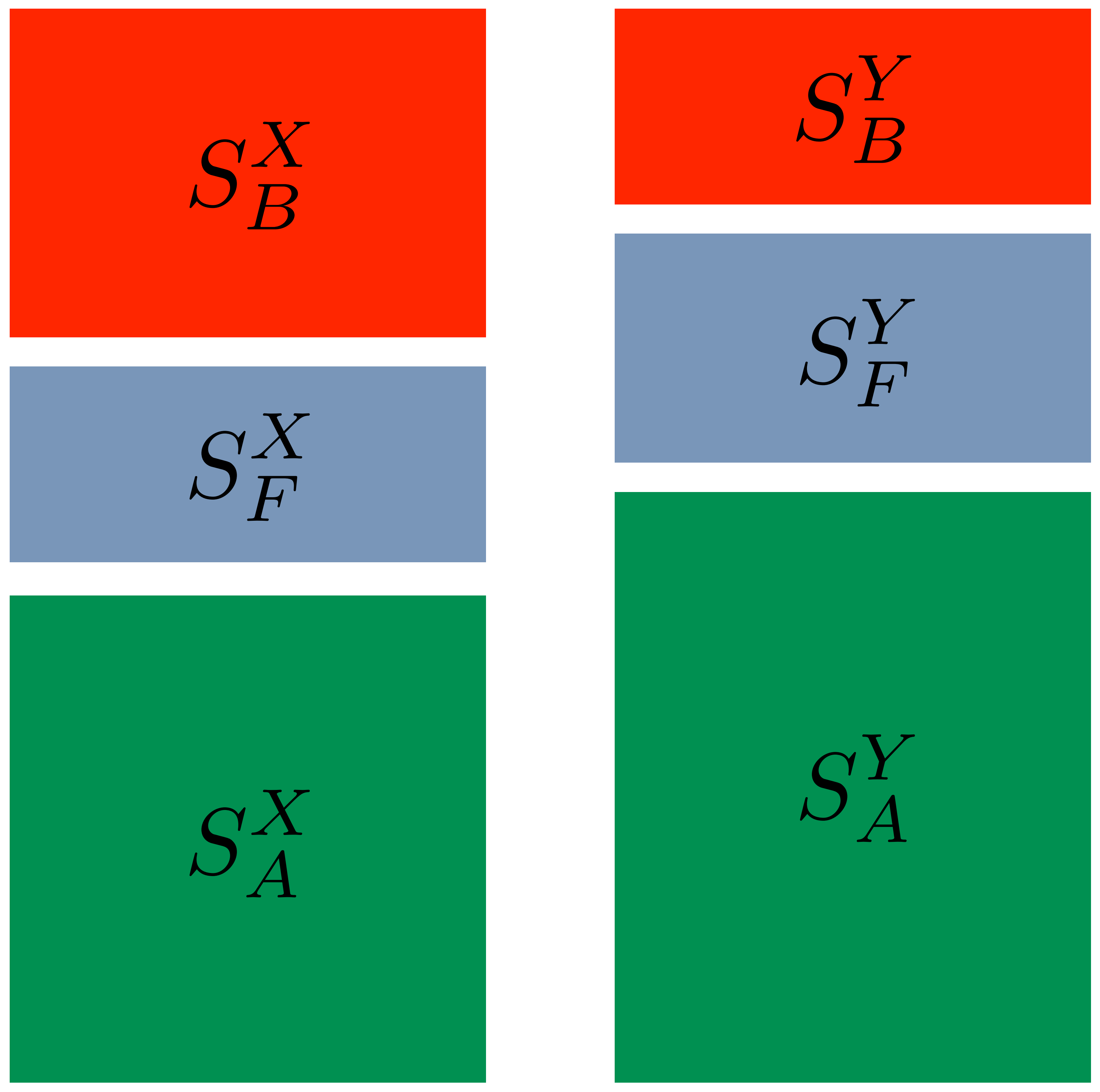
$$S_B^R = \{s_i \in S | R(s_i) > R(s_f), 1 \leq i \leq n\}$$
$$S_F^R = \{s_i \in S | R(s_i) = R(s_f), 1 \leq i \leq n\}$$
$$S_A^R = \{s_i \in S | R(s_i) < R(s_f), 1 \leq i \leq n\}$$

Crash Course into Our Proof System

- Formula R_1 *dominates* formula R_2 ($R_1 \rightarrow R_2$) if, for any combination of program, faulty statement, test suite, and a consistent tie-breaking scheme, $E_1 \leq E_2$
  - That is, $S_B^{R_1} \subseteq S_B^{R_2} \wedge S_A^{R_2} \subseteq S_A^{R_1}$
- Formula R_1 is *equivalent* to formula R_2 ($R_1 \leftrightarrow R_2$) if, for any combination of program, faulty statement, test suite, and a consistent tie-breaking scheme, $E_1 = E_2$
  - That is, $S_B^{R_1} = S_B^{R_1} \wedge S_F^{R_1} = S_F^{R_1} \wedge S_A^{R_1} = S_A^{R_1}$
- Finally, $R_1 \leftrightarrow R_2$ iff $R_1 \rightarrow R_2$ and $R_2 \rightarrow R_1$

Crash Course into Our Proof System

Statement Ranking



Formula X Formula Y

To show that Y dominates X,
we show that:

$$S_B^Y \subseteq S_B^X \wedge S_A^X \subseteq S_A^Y$$

(assuming that we break ties
in F sets consistently)

Equivalence is defined as:

$$X \leftrightarrow Y \iff X \rightarrow Y \wedge Y \rightarrow X$$

The Current Maximal Formulas

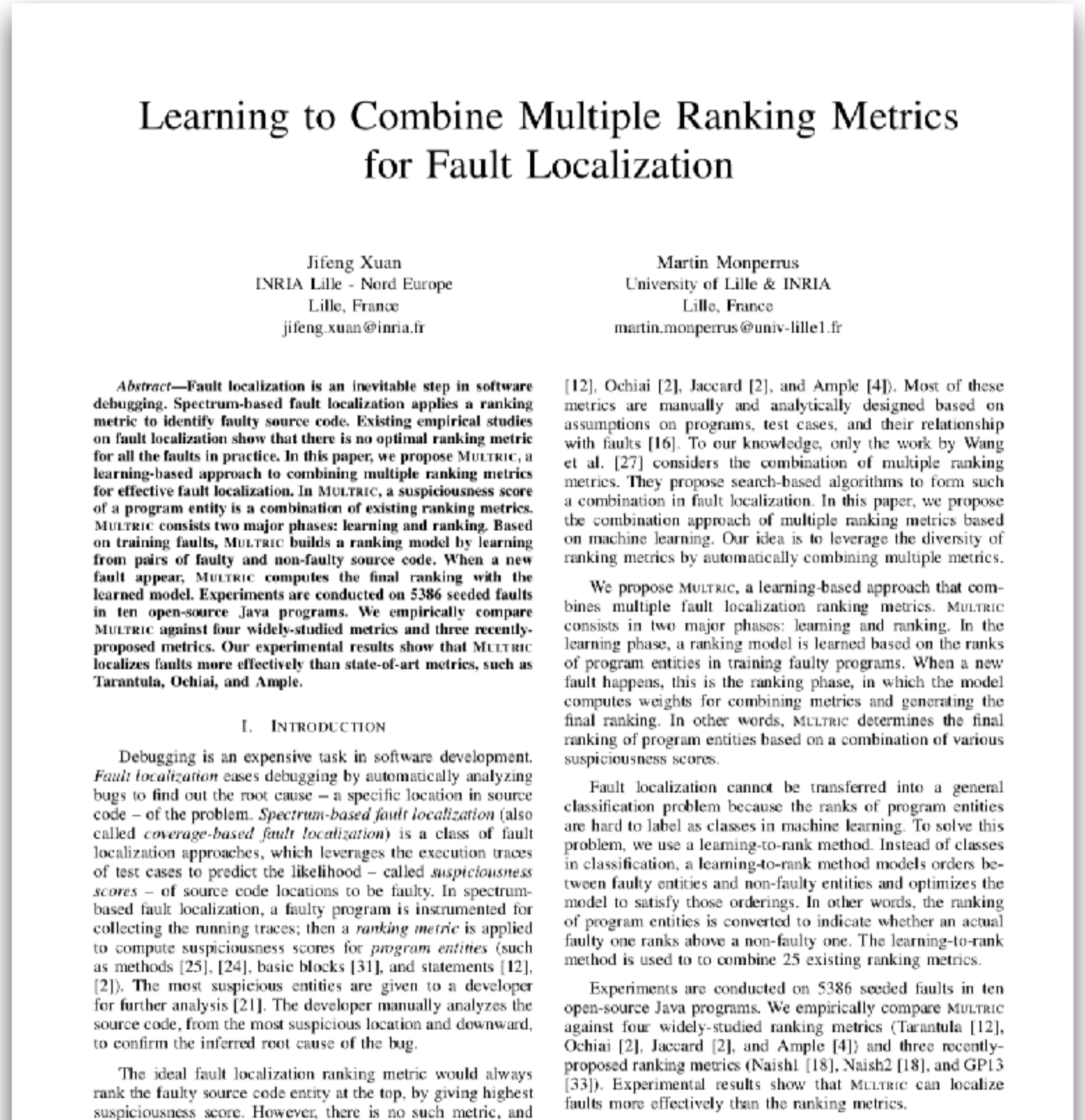
	Name	Formula expression
ER1’	Naish1	$\begin{cases} -1 & \text{if } e_f < F \\ P - e_p & \text{if } e_f = F \end{cases}$
	Naish2	$e_f - \frac{e_p}{e_p + n_p + 1}$
	GP13	$e_f(1 + \frac{1}{2e_p + e_f})$
ER5	Wong1	e_f
	Russel & Rao	$\frac{e_f}{e_f + n_f + e_p + n_p}$
	Binary	$\begin{cases} 0 & \text{if } e_f < F \\ 1 & \text{if } e_f = F \end{cases}$
GP02		$2(e_f + \sqrt{n_p}) + \sqrt{e_p}$
GP03		$\sqrt{\lvert e_f^2 - \sqrt{e_p} \rvert}$
GP19		$e_f \sqrt{\lvert e_p - e_f + n_f - n_p \rvert}$

Hybrid SBFL Approaches

- People are realising that a single formula is strongly limited.
- Hybridisation
  - Use multiple formulas at the same time, or
  - Use SBFL formulas with some additional input features
- As we accept more diverse input, fault localisation became a learning problem, instead of a design-a-technique problem

Xuan & Monperrus (ICSME 2014)

- Simultaneously use 25 SBFL formulas, using the weighted-sum method.
- Use a learning-to-rank technique to train the ranking model (i.e. to optimize the weights that yield best ranks for training set faults)



Learning to Combine Multiple Ranking Metrics for Fault Localization

Jifeng Xuan
INRIA Lille - Nord Europe
Lille, France
jifeng.xuan@inria.fr

Martin Monperrus
University of Lille & INRIA
Lille, France
martin.monperrus@univ-lille1.fr

Abstract—**Fault localization is an inevitable step in software debugging. Spectrum-based fault localization applies a ranking metric to identify faulty source code. Existing empirical studies on fault localization show that there is no optimal ranking metric for all the faults in practice. In this paper, we propose** MULTRIC**, a learning-based approach to combining multiple ranking metrics for effective fault localization. In** MULTRIC**, a suspiciousness score of a program entity is a combination of existing ranking metrics.** MULTRIC **consists two major phases: learning and ranking. Based on training faults,** MULTRIC **builds a ranking model by learning from pairs of faulty and non-faulty source code. When a new fault appear,** MULTRIC **computes the final ranking with the learned model. Experiments are conducted on 5386 seeded faults in ten open-source Java programs. We empirically compare** MULTRIC **against four widely-studied metrics and three recently-proposed metrics. Our experimental results show that** MULTRIC **localizes faults more effectively than state-of-art metrics, such as Tarantula, Ochiai, and Ample.**

I. INTRODUCTION

Debugging is an expensive task in software development. *Fault localization* eases debugging by automatically analyzing bugs to find out the root cause – a specific location in source code – of the problem. *Spectrum-based fault localization* (also called *coverage-based fault localization*) is a class of fault localization approaches, which leverages the execution traces of test cases to predict the likelihood – called *suspiciousness scores* – of source code locations to be faulty. In spectrum-based fault localization, a faulty program is instrumented for collecting the running traces; then a *ranking metric* is applied to compute suspiciousness scores for *program entities* (such as methods [25], [24], basic blocks [31], and statements [12], [2]). The most suspicious entities are given to a developer for further analysis [21]. The developer manually analyzes the source code, from the most suspicious location and downward, to confirm the inferred root cause of the bug.

The ideal fault localization ranking metric would always rank the faulty source code entity at the top, by giving highest suspiciousness score. However, there is no such metric, and

[12], Ochiai [2], Jaccard [2], and Ample [4]). Most of these metrics are manually and analytically designed based on assumptions on programs, test cases, and their relationship with faults [16]. To our knowledge, only the work by Wang et al. [27] considers the combination of multiple ranking metrics. They propose search-based algorithms to form such a combination in fault localization. In this paper, we propose the combination approach of multiple ranking metrics based on machine learning. Our idea is to leverage the diversity of ranking metrics by automatically combining multiple metrics.

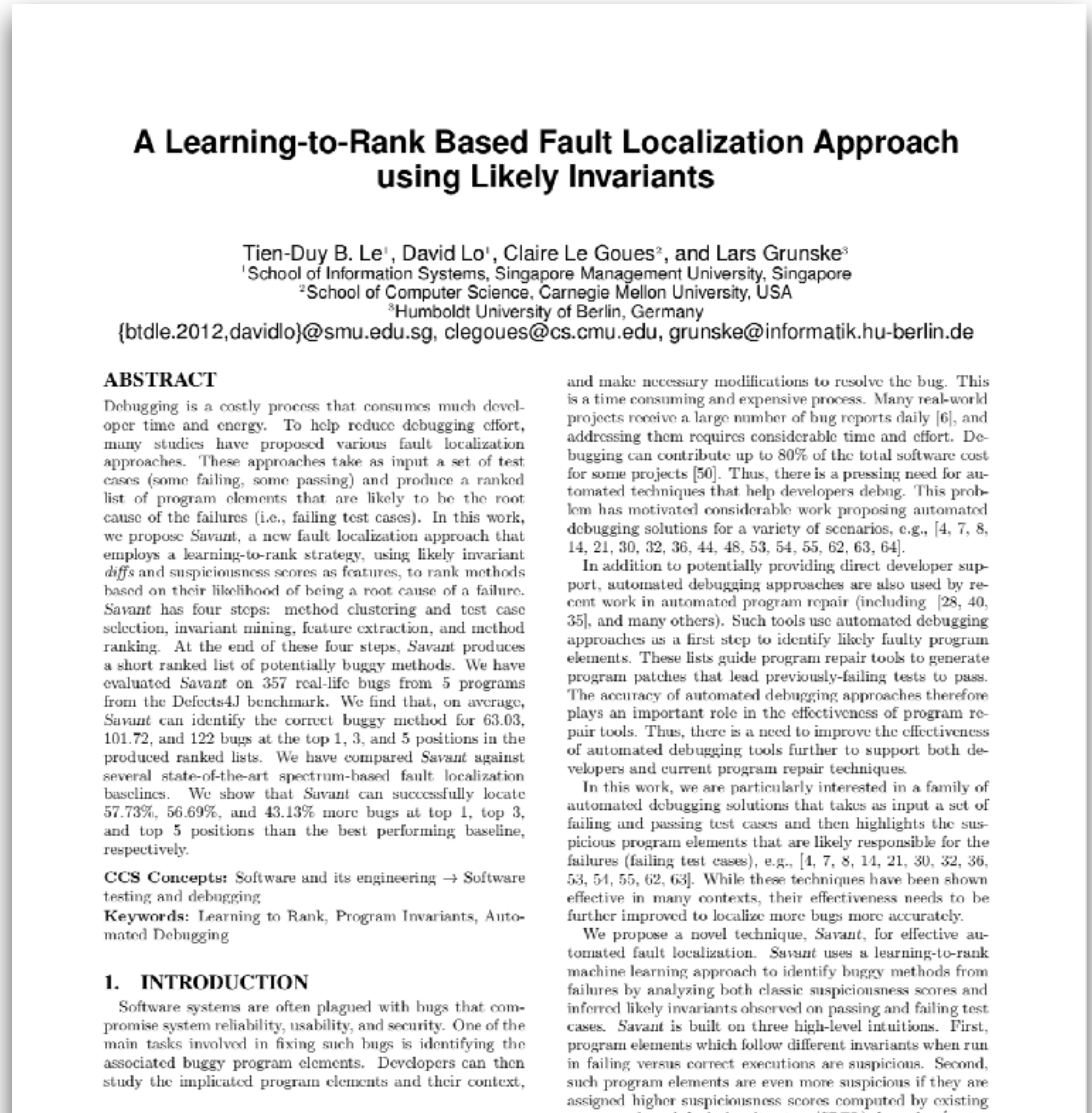
We propose MULTRIC, a learning-based approach that combines multiple fault localization ranking metrics. MULTRIC consists in two major phases: learning and ranking. In the learning phase, a ranking model is learned based on the ranks of program entities in training faulty programs. When a new fault happens, this is the ranking phase, in which the model computes weights for combining metrics and generating the final ranking. In other words, MULTRIC determines the final ranking of program entities based on a combination of various suspiciousness scores.

Fault localization cannot be transferred into a general classification problem because the ranks of program entities are hard to label as classes in machine learning. To solve this problem, we use a learning-to-rank method. Instead of classes in classification, a learning-to-rank method models orders between faulty entities and non-faulty entities and optimizes the model to satisfy those orderings. In other words, the ranking of program entities is converted to indicate whether an actual faulty one ranks above a non-faulty one. The learning-to-rank method is used to to combine 25 existing ranking metrics.

Experiments are conducted on 5386 seeded faults in ten open-source Java programs. We empirically compare MULTRIC against four widely-studied ranking metrics (Tarantula [12], Ochiai [2], Jaccard [2], and Ample [4]) and three recently-proposed ranking metrics (Naish1 [18], Naish2 [18], and GP13 [33]). Experimental results show that MULTRIC can localize faults more effectively than the ranking metrics.

Le et al. (ISSTA 2016)

- In addition to 35 SBFL formulas, *Savant* use invariant difference:
  - Use Daikon to infer method invariants twice: with passing tests, and with failing test. For the faulty method, two sets of invariant will tend to be more different.



A Learning-to-Rank Based Fault Localization Approach using Likely Invariants

Tien-Duy B. Le[1], David Lo[1], Claire Le Goues[2], and Lars Grunske[3]
[1]School of Information Systems, Singapore Management University, Singapore
[2]School of Computer Science, Carnegie Mellon University, USA
[3]Humboldt University of Berlin, Germany
{btdle.2012,davidlo}@smu.edu.sg, clegoues@cs.cmu.edu, grunske@informatik.hu-berlin.de

ABSTRACT

Debugging is a costly process that consumes much developer time and energy. To help reduce debugging effort, many studies have proposed various fault localization approaches. These approaches take as input a set of test cases (some failing, some passing) and produce a ranked list of program elements that are likely to be the root cause of the failures (i.e., failing test cases). In this work, we propose *Savant*, a new fault localization approach that employs a learning-to-rank strategy, using likely invariant *diffs* and suspiciousness scores as features, to rank methods based on their likelihood of being a root cause of a failure. *Savant* has four steps: method clustering and test case selection, invariant mining, feature extraction, and method ranking. At the end of these four steps, *Savant* produces a short ranked list of potentially buggy methods. We have evaluated *Savant* on 357 real-life bugs from 5 programs from the Defects4J benchmark. We find that, on average, *Savant* can identify the correct buggy method for 63.03, 101.72, and 122 bugs at the top 1, 3, and 5 positions in the produced ranked lists. We have compared *Savant* against several state-of-the-art spectrum-based fault localization baselines. We show that *Savant* can successfully locate 57.73%, 56.69%, and 43.13% more bugs at top 1, top 3, and top 5 positions than the best performing baseline, respectively.

CCS Concepts: Software and its engineering → Software testing and debugging

Keywords: Learning to Rank, Program Invariants, Automated Debugging

1. INTRODUCTION

Software systems are often plagued with bugs that compromise system reliability, usability, and security. One of the main tasks involved in fixing such bugs is identifying the associated buggy program elements. Developers can then study the implicated program elements and their context, and make necessary modifications to resolve the bug. This is a time consuming and expensive process. Many real-world projects receive a large number of bug reports daily [6], and addressing them requires considerable time and effort. Debugging can contribute up to 80% of the total software cost for some projects [50]. Thus, there is a pressing need for automated techniques that help developers debug. This problem has motivated considerable work proposing automated debugging solutions for a variety of scenarios, e.g., [4, 7, 8, 14, 21, 30, 32, 36, 44, 48, 53, 54, 55, 62, 63, 64].

In addition to potentially providing direct developer support, automated debugging approaches are also used by recent work in automated program repair (including [28, 40, 35], and many others). Such tools use automated debugging approaches as a first step to identify likely faulty program elements. These lists guide program repair tools to generate program patches that lead previously-failing tests to pass. The accuracy of automated debugging approaches therefore plays an important role in the effectiveness of program repair tools. Thus, there is a need to improve the effectiveness of automated debugging tools further to support both developers and current program repair techniques.

In this work, we are particularly interested in a family of automated debugging solutions that takes as input a set of failing and passing test cases and then highlights the suspicious program elements that are likely responsible for the failures (failing test cases), e.g., [4, 7, 8, 14, 21, 30, 32, 36, 53, 54, 55, 62, 63]. While these techniques have been shown effective in many contexts, their effectiveness needs to be further improved to localize more bugs more accurately.

We propose a novel technique, *Savant*, for effective automated fault localization. *Savant* uses a learning-to-rank machine learning approach to identify buggy methods from failures by analyzing both classic suspiciousness scores and inferred likely invariants observed on passing and failing test cases. *Savant* is built on three high-level intuitions. First, program elements which follow different invariants when run in failing versus correct executions are suspicious. Second, such program elements are even more suspicious if they are assigned higher suspiciousness scores computed by existing

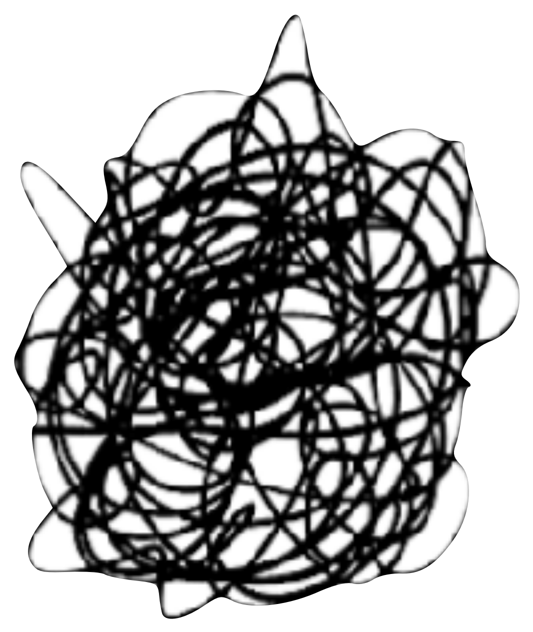
Sohn & Yoo (ISSTA 2017)

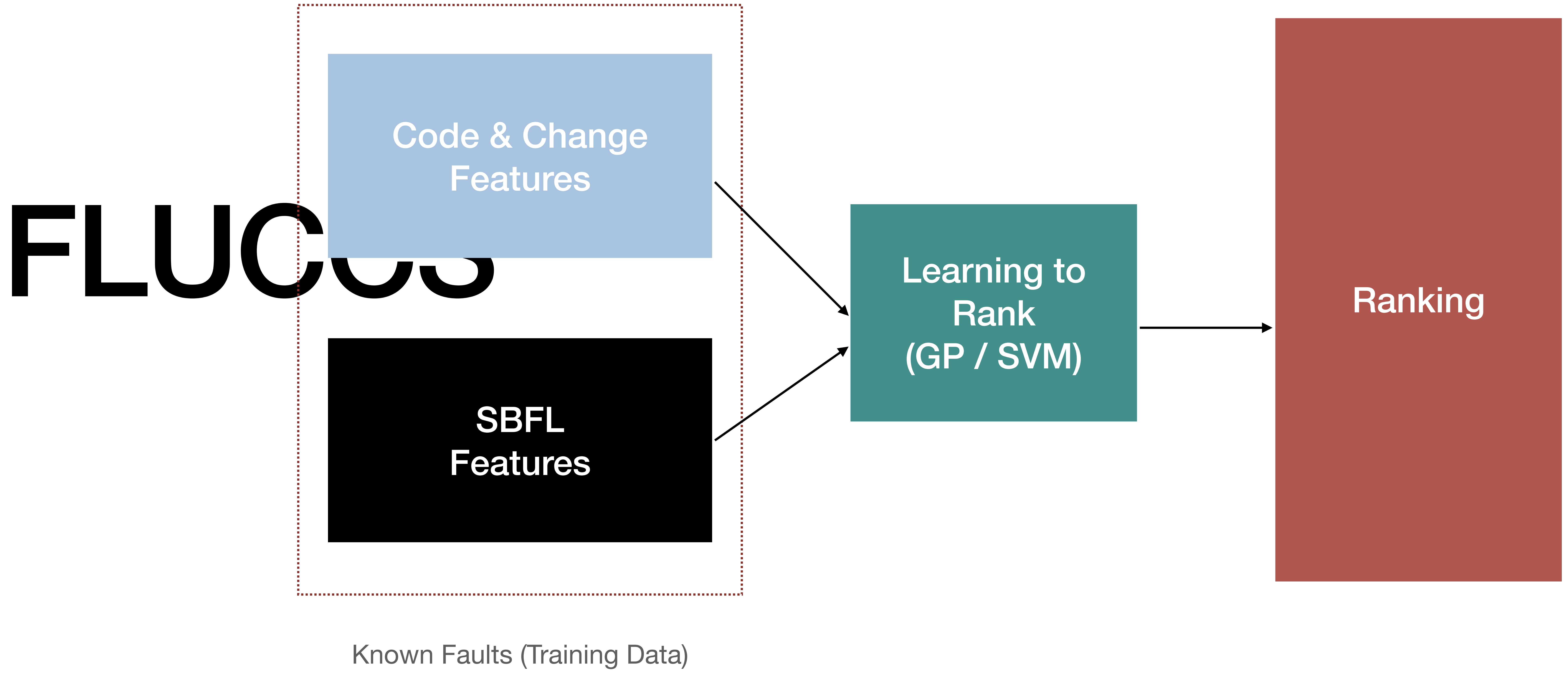
- A simple and naive insight: we already have a technique that claims it can tell us where the faults are - even before we test!
- 😰
- But, seriously, we present FLUCCS - Fault Localisation Using Code and Change Metrics.

DEFECT PREDICTION

FLUCCS: Code and Change Metrics Features

- Age: how long has the given element existed in the code base?
- Churn: how frequently has the given element been changed?
- Complexity: how complex is the given element




FLUCCS
Code & Change Features
SBFL Features
Known Faults (Training Data)
Learning to Rank (GP / SVM)
Ranking

SBFL

- Strengths
    - Only requires what is already there: coverage and test results
    - Relatively intuitive
- Weaknesses
    - Single formulas are usually limited. In fact,
    - There exists a theoretical proof that no single formula works the best against all faults
    - Does not work against omission faults
    - Does not work well against multiple faults

Mutation Based Fault Localisation

- How do we use mutants to localise a fault? Mutants are injected faults themselves!!
- Consider this: what would happen if you mutate a program you know to be faulty?

Case 1: Mutating Correct Statements

New Fault

P- F+

Equivalent

P F

Mask

P+ F-

CASE 2: mutating Faulty Statement

(Partial) Fix

P+ F-

(New) Fault

P? F?

Equivalent

P F

Mask

P+ F-

Hypotheses

- An arbitrary mutation operator applied to a correct statement is likely to introduce a new fault
- An arbitrary mutation operator applied to a faulty statement is either likely to keep the program still faulty or, even better, (partially) fix the program
- The majority of statements in a faulty program is correct; we detect the faulty one by observing the anomaly from our hypotheses

MUSE (Moon et al., 2014)

Proportion of test cases that mutant m turns from fail to pass

$$\mu(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|f_P(s) \cap p_m|}{|f_P|} - \alpha \cdot \frac{|p_P(s) \cap f_m|}{|p_P|} \right)$$

Average over all mutation applied to statement s

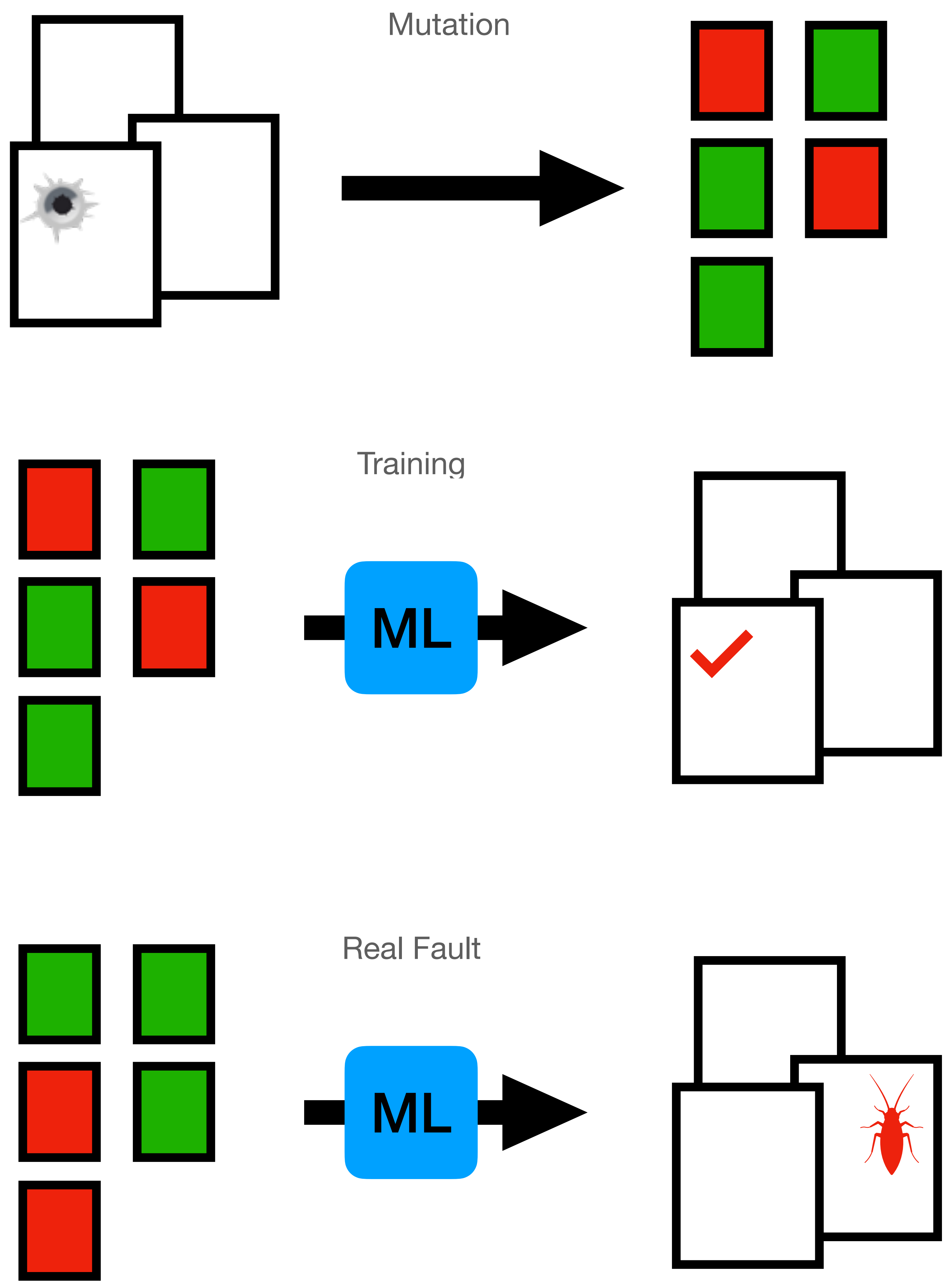
Proportion of test cases that mutant m turns from pass to fail

$$\alpha = \frac{f2p}{|mut(P)| \cdot |f_P|} \cdot \frac{|mut(P)| \cdot |p_P|}{p2f}$$

Ahead of Time MBFL

Seshat (Kim et al., ISSRE 2021)

- Perform mutation analysis (i.e., inject faults into different locations, and record which tests fail)
- Using this information, learn the reverse relationship. The ML model is designed to answer this question: “if these test cases failed, where is the mutant?”
- When real faults occur, give the test outcome to the trained model, and ask where the mutant (=real fault) is.
- Promising results (113 out of 203 faults ranked at the top, method level FL on Java)



Summary

- Delta Debugging minimises a failure-inducing input, thereby helping the developer to localise the corresponding part in the program.
- IR based FL queries the documents (source code files) using symptoms of the failure (bug report) as the query text.
- SBFL capsulate the intuition about test results and coverage into risk evaluation formula.
- MBFL is based on the idea that mutating an already faulty program can reveal insights about the fault (further damage or partial fix).