Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization*

Xinming Wang
Dept. of Comp. Sci. & Eng.
HKUST
Hong Kong, China
rubin@cse.ust.hk

S.C. Cheung[§]
Dept. of Comp. Sci. & Eng.
HKUST
Hong Kong, China
scc@cse.ust.hk

W.K. Chan
Dept. of Comp. Sci.
City Univ. of Hong Kong
Hong Kong, China
wkchan@cs.cityu.edu.hk

Zhenyu Zhang
Dept. of Comp. Sci.
HKU
Hong Kong, China
zyzhang@cs.hku.hk

Abstract

Recent techniques for fault localization leverage code coverage to address the high cost problem of debugging. These techniques exploit the correlations between program failures and the coverage of program entities as the clue in locating faults. Experimental evidence shows that the effectiveness of these techniques can be affected adversely by coincidental correctness, which occurs when a fault is executed but no failure is detected. In this paper, we propose an approach to address this problem. We refine code coverage of test runs using control- and data-flow patterns prescribed by different fault types. We conjecture that this extra information, which we call context patterns, can strengthen the correlations between program failures and the coverage of faulty program entities, making it easier for fault localization techniques to locate the faults. To evaluate the proposed approach, we have conducted a mutation analysis on three real world programs and cross-validated the results with real faults. The experimental results consistently show that coverage refinement is effective in easing the coincidental correctness problem in fault localization techniques.

1. Introduction

Debugging is a tedious and expensive activity in software maintenance. As a major part in debugging, fault localization consumes the most amounts of time and effort [27]. To reduce the expense of fault localization, researchers have proposed automatic fault localization techniques. Among them, one promising approach is to locate faults using code coverage information, that is, the set of program entities (e.g., statements or branches) executed in each test run. This approach is generally referred to as *coverage-based fault localization* (CBFL) [29]. Examples of CBFL techniques include Tarantula [18], Ochiai [1], and χDebug [31]. Studies [18][19] show that these techniques can be effective in finding faults in programs.

In general, CBFL techniques collect code coverage from both failed test runs (each of which detects a failure) and passed test runs (where no program failure is detected). They then search for program entities whose coverage strongly correlates with program failures. These program entities are regarded as the likely faulty locations.

Despite the correlation-based fault localization strategy of CBFL has delivered promising results in previous experiments (e.g., [5][19][21]), in practice its effectiveness can be affected by many factors. One such factor is coincidental correctness, which occurs when "*no failure is detected, even though a fault has been executed*" [26]. Previously, coincidental correctness has been perceived as a problem and attracted many research interests (e.g., [14][15][23][26]) because studies show that it can adversely affect the effectiveness of testing [11]. Recent experimental evidence shows that it is undesirable to CBFL techniques as well. For example, Jones and colleagues [18] noticed that Tarantula fails to highlight faulty statements in the initialization code or main program path (e.g., code in the "main" function of a C program). They suggested that coincidental correctness may be the culprit and called for further investigation. Such adverse cases were also found by other researchers (e.g., [5][32]). In our experiment (reported in Section 6), we observed that the effectiveness of CBFL declines significantly when the occurrence of coincidental correctness increases. These observations are of concern because coincidental correctness can be very common (as shown in empirical studies on coverage testing [17] and our experiment).

The goal of this work is to reduce the vulnerability of CBFL to coincidental correctness. This goal is challenging because we do not know where the faults reside and have no way of directly identifying passed test runs where coincidental correctness has occurred. To address this challenge, our approach is to transform the code coverage in a way that will strengthen the correlations between program failures and the coverage of faulty program entities. We refer to such an approach as *coverage refinement*.

Our approach is inspired by backward (dynamic) slicing [2], which is a possible way of coverage refinement.

[*] This work was partially supported by the Research Grants Council of Hong Kong under grant numbers 612108, 111107, 123207; and RPC07/08.EG24.

[§] Correspondence author.

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

The idea is to exclude the coverage of those program entities whose execution does not affect the output. The intuition is that a faulty program entity cannot trigger the failure unless the output is dynamically dependent on its execution. This intuition is valid for faults involving wrong arithmetic or Boolean expressions. However, it is invalid for many others. For example, backward slicing cannot handle faults involving code omission [33]. Such faults are arguably more difficult to debug [33] and more common in practice than faults related to wrong code construct (e.g., 64% vs. 33% [13]).

This suggests that coverage refinement using backward slicing alone is inadequate. To address this problem, we observe that when the execution of faults triggers the failure, the dynamic control flows and data flows before and after their execution usually match certain patterns. We refer to these patterns as *context patterns*. Indeed, coverage refinement with backward slicing exploits one such pattern that features the existence of a dynamic program dependence chain from the fault execution to the output. However, this context pattern is invalid for other types of faults, notably those related to code omission. We conjecture that context patterns for common fault types can be derived and used for coverage refinement purpose.

To validate our conjecture, we have conducted a case study on the fault types identified in recent field study [13] (dominated by code omission faults) and derived context patterns for each of them. To investigate how effectively the coverage refinement approach addresses the coincidental correctness problem in CBFL, we conducted a mutation analysis [4] on three real-world programs with these fault types and context patterns involved in the case study. For cross validation purpose, we also repeated the experiment with real faults. The results consistently show that the use of context patterns is highly effective in addressing the coincidental correctness problem and reducing the effort programmers spent on fault localization.

This paper makes the following main contributions:

1) The introduction of context patterns to refine code coverage, alleviating the coincidental correctness problem in CBFL.
2) A case study that investigates the feasibility of deriving context patterns for common fault types.
3) Empirical investigation on the effects of coincidental correctness upon CBFL, and how effective coverage refinement with context patterns addresses this problem.

The rest of this paper is organized as follows. Section 2 summarizes automatic fault localization techniques. Section 3 uses an example to discuss the coincidental correctness problem and introduces our approach of coverage refinement with context patterns. Section 4 presents how to specify and match context patterns. Section 5 and 6 report the evaluation results. Finally, Section 7 concludes.

2. Background and related work

Coverage-Based Fault Localization Techniques

Many CBFL techniques have been proposed. They take code coverage as input and produce a set of likely faulty program entities as output. In principle, any kind of code coverage can be used. However, in the literature they are mainly applied to statement coverage. We follow this convention and use statements as program entities.

Agrawal and colleagues [3] are among the first to use code coverage for automatic fault localization purpose. Their technique, called χSlice, collects coverage from a failed test run and a passed test run. The *dice* of them, that is, the set of statements executed only in the failed test run, is reported as the likely faulty statements. Therefore, their fault localization strategy is to search for statements whose coverage *strictly* correlates with program failures. This idea is further developed by Renieris and Reiss [25]. Their technique, called Nearest Neighborhood (NN), features an extra step of passed test run selection.

Jones and colleagues [18] proposed a different CBFL technique called Tarantula. Unlike that of χSlice, the fault localization strategy of Tarantula is to search for statements whose coverage has a relatively strong (but not necessarily strict) correlation with program failures. Tarantula defines a color scheme to measure the correlation. Since we do not use visualization here, we find it more appropriate to rename the measurement as *failure-correlation*. For each statement *S* executed in at least one test run, this measurement is defined as:

$$\textit{failure-correlation}(S) = \frac{\%failed(S)}{\%failed(S) + \%passed(S)} \quad \text{(F1)}$$

, where *%failed*(*S*) is the percentage of failed test runs executing *S*, and *%passed*(*S*) is percentage of passed test runs executing *S*. The value of failure-correlation ranges from 0 to 1. Higher failure-correlation value suggests that *S* is more likely to be faulty. When two statements have the same failure-correlation value, another measurement:

$$\textit{Confidence} = \textit{max}(\%\textit{failed}(S), \%\textit{passed}(S)) \quad \text{(F2)}$$

is used as a tie-breaker. Jones and Harrold [19] empirically compared Tarantula with χSlice and NN. Their result shows that Tarantula performs the best among them.

Recently, researchers have proposed new CBFL techniques, such as Ochiai [1] and χDebug [31]. These techniques are similar to Tarantula except that they use different formulas to compute failure-correlation (see [29] for a survey). As Tarantula is representative, we use it in our discussion for the rest of this work.

Other Automatic Fault Localization Approaches

Statistical debugging [21][22] instruments predicates in the program (for examples, the comparison between two variables or the sign of function return value) and locates faults by comparing the evaluation results of predicates in failed test runs with those in all test runs. In certain sense, predicates can be regarded as another way of coverage refinement by exploiting the program state information. This is different from our approach, which use control flow and data flow information. We note, however, that these two approaches are complementary and can be combined. In the future, we shall conduct studies to investigate whether such a combination can further improve the effectiveness of CBFL.

Delta debugging [8][34] grafts values from a failed test run to a passed test run. This is systematically tried for different program locations and variables. When one such trial duplicates the failure observed in the failed test run, the variable and the program location under scrutiny could help locating faults. Delta debugging has been shown to be useful in revealing many real world faults. However, the cost of repeating the trials can be expensive [33]. Besides, existing experiment results (e.g., [19]) show that when multiple test runs are available, the performance of CBFL is better than that of delta debugging.

3. Research problem and our approach

In this section, we use a working example to discuss the coincidental correctness problem in CBFL, and motivate the concepts of coverage refinement and context pattern. Formal treatment of them is given in Section 4.

Figure 1(a) shows a program with an assignment statement intentionally commented out to simulate a missing assignment fault. This kind of faults related to code omission is common [13]. For this kind of faults, we follow the convention adopted by Jones and Harrold [18], and consider the statement preceding the missing one (S_0 in this example) to be faulty because this statement would direct programmers attention to the omission place.

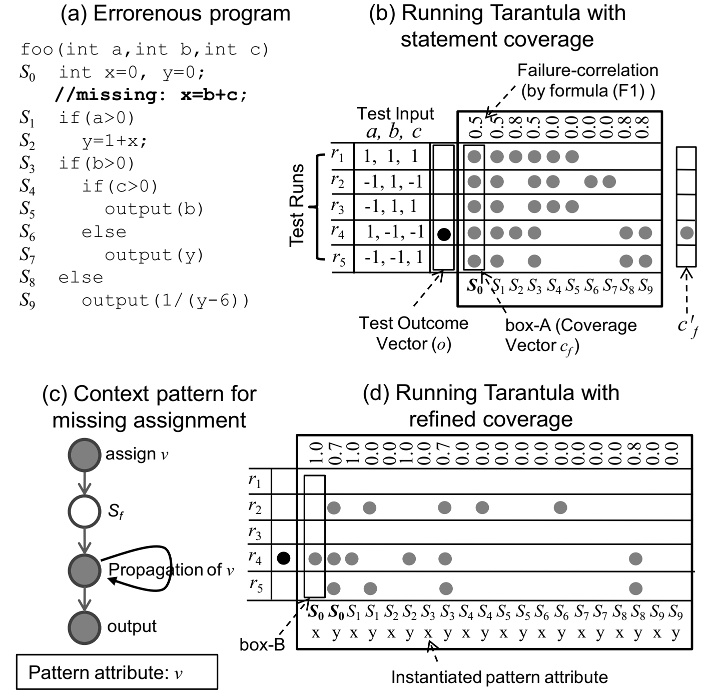
Coincidental Correctness Problem in CBFL

Suppose that programmers use Tarantula to locate the fault in this program. To collect coverage, this program is executed by five test runs, each of which corresponds to one row in Figure 1(b). For each test run, we give its input, test outcome, and statement coverage. Each column can be regarded as a 0-1 vector with dots representing ‘1’s. The ‘1’s in the test outcome vector represent failures, while those in coverage vectors indicate that the corresponding statement is executed in the test run.

From Figure 1(b), we observe that coincidental correctness occurs in all passed test runs. By formula (F1), the faulty statement S_0 will be assigned with a medium failure-correlation value 0.5 (box A). As half of the statements in the program have the same or higher failure-correlation value, S_0 has not been accurately isolated.

Coverage Refinement and Context Patterns

For clarity in what follows, we denote the faulty statement as S_f, the test outcome vector as o, and the coverage vector of S_f as c_f.



(a) Errorenous program
(b) Running Tarantula with statement coverage
Failure-correlation (by formula (F1))
Test Input
Test Runs
Test Outcome Vector (o)
box-A (Coverage Vector c_f)
c'_f
(c) Context pattern for missing assignment
assign v
S_f
Propagation of v
output
Pattern attribute: v
(d) Running Tarantula with refined coverage
box-B
Instantiated pattern attribute

```
    foo(int a,int b,int c)
S0    int x=0, y=0;
      //missing: x=b+c;
S1    if(a>0)
S2      y=1+x;
S3    if(b>0)
S4      if(c>0)
S5        output(b)
S6      else
S7        output(y)
S8    else
S9      output(1/(y-6))
```

	a, b, c	o	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	c'_f
			0.5	0.5	0.8	0.5	0.0	0.0	0.0	0.0	0.8	0.8	
r_1	1, 1, 1		●	●	●	●	●	●					
r_2	-1, 1, -1		●	●		●	●		●	●			
r_3	-1, 1, 1		●	●		●	●	●					
r_4	1, -1, -1	●	●	●	●	●					●	●	●
r_5	-1, -1, 1		●	●		●					●	●	

		S_0	S_0	S_1	S_1	S_2	S_2	S_3	S_3	S_4	S_4	S_5	S_5	S_6	S_6	S_7	S_7	S_8	S_8	S_9	S_9
		x	y	x	y	x	y	x	y	x	y	x	y	x	y	x	y	x	y	x	y
		1.0	0.7	1.0	0.0	0.0	1.0	0.0	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.8	0.0	0.0
r_1																					
r_2			●		●				●		●				●						
r_3																					
r_4	●	●	●	●			●		●										●		
r_5			●		●				●										●		

Figure 1: Example illustrating the coincidental correctness problem in CBFL and coverage refinement

The example in Figure 1(a) shows that the occurrence of the failure depends not only on the execution of S_f, but also on the control flows and data flows before and after the execution of S_f. We call the latter factor the *execution context* of S_f.

Now suppose that p is a pattern of execution context that satisfies the following two criteria:

- If S_f has been executed in a test run r but its execution did not trigger the failure, then p is never matched by the execution contexts of S_f in r.
- If S_f has been executed in a test run r and its execution triggered the failure, then p is matched at least once by the execution contexts of S_f in r.

Then, by removing the ‘1’s in c_f that correspond to test runs where S_f’s execution contexts never match p, we thus transform c_f into another vector c_f' that is identical to o (also a vector). With this refined coverage vector, S_f can be more accurately isolated because by formula (F1), it will have the maximal failure-correlation value 1.0.

In applying this basic idea, there are three issues:

1) *How to obtain the context pattern p?* At this point, let us suppose that we know the type of faults in the program. Then one approach to deriving p is to capture the mechanism of how faults of this type trigger the failures. Such mechanism for different fault types has been extensively studied both analytically (e.g., the RELAY model [26]) and empirically (e.g., [11]).

For example, missing assignment triggers the failure only when the obsolete value propagates to the output [28]. This mechanism is captured by the pattern shown in Figure 1(c). In this figure, nodes represent statements and edges represent "executed before" relations. This pattern has an attribute *v*, which corresponds to the variable whose assignment is absent.

Note that patterns derived in this way might not strictly satisfy the above-mentioned two criteria. For example, when the absent assignment is redundant, the execution of S_f may not trigger any failure, yet the pattern shown in Figure 1(c) is still matched. Coverage refinement can still be effective at increasing the failure correlation of S_f if the pattern satisfies the two criteria in most of the cases. However, this conjecture needs empirical evaluation.

2) *S_f and c_f are unknown*. As the faulty statement S_f is unknown, our approach refines the coverage vectors of all statements with the context pattern *p*. In doing so, we need to validate the assumption that *p* acts on non-faulty statements in a random fashion, and few of them will have their failure-correlation values dramatically increased.

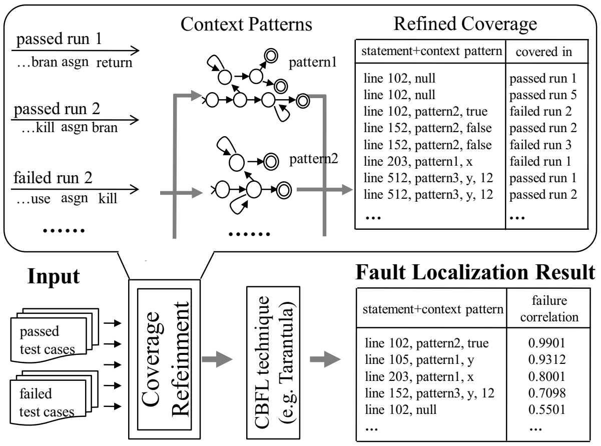
Figure 1(d) shows the coverage of S_0-S_9 refined with the pattern depicted in Figure 1(c). A statement now spans multiple coverage vectors, each of which corresponds to a context pattern matched on its execution context with an instantiation of pattern attributes. As shown in the figure, the faulty statement S_0 (box B) is one of the three statements that have the maximal failure correlation value 1.0. Therefore, S_0 is more accurately isolated.

3) *The types of fault in the program are unknown*. In the above discussion, we derive the context pattern based on the fault type information. However, in reality the types of fault in the program are unknown. A related issue is that the program might contain multiple types of fault.

While the precise fault types in a specific faulty program are unknown, programmers might still have information on the range of possible fault types. This information can come from field studies (e.g., [13]) or experiences on the project development history (e.g., [20]). Our approach assumes the availability of fault types from these sources and refines the coverage with their corresponding context patterns simultaneously. Of course, the conjectured fault types do not necessarily include the real fault types. To avoid misleading the programmers by decreasing the failure-correlation of faulty statements, we can enlist a "null" pattern that matches any execution context.

Review and Research Questions

Figure 2 summarizes the above discussion and gives the overview of our approach. In our approach, test runs are modeled as sequences of control flow and data flow events. Context patterns are matched against these sequences (Section 4 discusses the details of context pattern definition and matching). The matching results are combined into refined coverage, which is the same as statement coverage except that the coverage unit changes from "statement" to "statement + context pattern". Finally, CBFL techniques work on this refined coverage and produce the fault localization result.



statement+context pattern	covered in
line 102, null	passed run 1
line 102, null	passed run 5
line 102, pattern2, true	failed run 2
line 152, pattern2, false	passed run 2
line 152, pattern2, false	failed run 3
line 203, pattern1, x	failed run 1
line 512, pattern3, y, 12	passed run 1
line 512, pattern3, y, 12	passed run 2
…	…

statement+context pattern	failure correlation
line 102, pattern2, true	0.9901
line 105, pattern1, y	0.9312
line 203, pattern1, x	0.8001
line 152, pattern3, y, 12	0.7098
line 102, null	0.5501
…	…

Figure 2: Overview of our approach

Having introduced our approach, we next consider research questions regarding to its practical implication:

RQ1: Is there a way to derive effective context patterns for common fault types? In above discussion, we outlined an approach to deriving context patterns by capturing the mechanism of how faults trigger failures. Whether context patterns can capture such mechanisms for common fault types should be investigated in a case study.

RQ2: How severe is the coincidental correctness problem in CBFL? While researchers have found adverse cases (e.g., [5][18]), it is still unclear whether such cases are common. Investigation of this issue helps understand the practical significance of our coverage refinement approach. Especially, we are interested in *how often coincidental correctness occurs (RQ2.1*), and *how severely its occurrence affects the effectiveness of CBFL* (*RQ2.2*).

RQ3: How effectively does coverage refinement help isolating faulty statements when coincidental correctness occurs frequently? Despite coverage refinement can increase the failure correlation of faulty statements, it might increase that of non-faulty statements at the same time. Coverage refinement is helpful only when the former effect supersedes the latter effect (as we hope). Therefore, the benefit of coverage refinement is not immediately obvious and needs empirical investigation.

In Section 5, we investigate RQ1 in a case study. In Section 6, we investigate RQ2 and RQ3 through controlled experiments. To lay the necessary foundation for these investigations, we describe how we specify and match context patterns in the next section.

4. Specifying and matching context patterns

In this section, we first briefly summarize event expression [6], and then present how we use it to specify context patterns and its matching algorithm.

4.1 Preliminary: event expression

A context pattern matches a sequence of control flow or data flow events with specific relations. In our study,

```
PATTERN  asgn x, (use y)+, output z
SATISFY  first(y).dep_stmt == x.stmt &&
         next(y).dep_stmt == prev(y).stmt,
         z.stmt == last(y).stmt
```

Figure 3: An example of event expression

we choose event expression to describe context patterns, because event expression is formal and simple in syntax. Furthermore, it is capable of describing a wide variety of patterns [6]. In the following, we briefly introduce its syntax. Its semantics is in Appendix A.

Figure 3 shows an event expression that matches the propagation of an assignment to the output. Each event expression consists of two clauses. The **PATTERN** clause is similar to a regular expression. It consists of several pairs of event type and event variable connected by one of the following operators:

- `A,B` and `A;B` (Sequential): `A` and `B` are matched on the event sequence in tandem. "`A;B`" requires that their matches are contiguous, while "`A,B`" does not.
- `A&B` (Concurrency): `A` and `B` are matched independently on the event sequence.
- `A|B` (Choice): either `A` or `B` is matched on the event sequence.
- `A+` (Kleene closure): `A` is matched on the event sequence at least once.
- `~A` (Negation): `A` is not matched on the event sequence.

Next, the **SATISFY** clause specifies the relation between attributes of event variables. It is essentially a Boolean expression with common relational and logic operators used in C. Event variables that appear in a Kleene closure, such as `y` in Figure 3, can be applied with several built-in functions. Take `y` for example, the possible functions and their meanings are:

- `y`: the whole closure.
- `first(y)`: the first event in the closure.
- `last(y)`: the last event in the closure.
- `next(y)`: the whole closure except for the first event.
- `prev(y)`: the whole closure except for the last event.
- `len(y)`: the number of events in the closure.

The above functions can be used with relational operators. For example, the expression `next(y).dep_stmt ==prev(y).stmt` means that the attribute *dep_stmt* of the *i*-th event in `next(y)` is equal to the attribute *stmt* of the *i*-th events in `prev(y)`, where $i \in$[1, `len(y)`-1].

4.2 Context pattern description

Besides event expression, the description of a context pattern requires an event model of program execution and directives for the generation of refined coverage.

Event model: To evaluate our approach, we have defined a compact event model for C programs (Table 1), which captures the essential dynamic control flow and data flow information. This model is not intended to describe all aspects of program execution. Yet it allows us to specify many interesting context patterns.

Table 1: The event model for our study

Type	Attributes	Description
asgn	stmt, func, var_name, value, asgn_type:{formal_in, formal_out, local, global}	A variable (*var_name*) is assigned with a value.
use	stmt, func, type:{asgn, bran, output}, dep_stmt, dep_type:{data, control},	*stmt* is dynamic control or data dependent on *dep_stmt*.
kill	stmt, func, var_name	A variable is overwritten, or it leaves the scope.
bran	stmt, func, direction:{T, F}	A branching is taken.
bran_exit	stmt, func	A branch is left.
entry, return	stmt, func, caller, callsite	A function is entered or left
call_lib_ func	stmt, func, lib_func_name, handle	A library function is called with *handle* as argument.
output	stmt, func, var_name	A variable is outputted.

In this event model, the execution of a statement is decomposed into one or more events with different types. For example, the execution of an assignment is decomposed into zero or more events with type *use*, followed by zero or more events with type *kill*, and then followed by one event with type *asgn*. Each event has a fixed set of attributes. Two common attributes *stmt* and *func* are contained by every event type. The attribute *stmt* specifies the statement execution instance that generates this event, and the attribute *func* specifies the function execution instance in which the event occurs. Other event attributes specify details of the statement execution and have straightforward meanings.

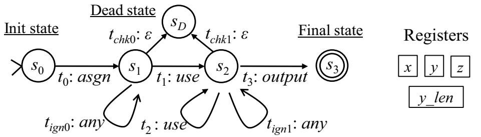
Refined coverage generation directives: We add to event expression two simple clauses to specify how to generate the refined coverage when the pattern is matched. The **REFINE** clause specifies the line number of the statement whose coverage is refined. The **ATTRS** clause specifies the pattern attributes.

4.3 Matching context patterns

Given an event expression, we translate it into an *extended finite state machine* (EFSM) [7] and then execute this EFSM on event sequences. EFSM is an extension of finite state machine (FSM) with three enhancements.

- A set of *registers* representing the "external state" of the finite state machine.
- A set of *enabling conditions*, each of which labels a transition and specifies the condition under which this transition is allowed to fire.
- A set of *update actions*, each of which labels a transition and specifies how registers are updated when this transition is fired.

Figure 4 illustrates the EFSM translated from the event



trans	enabling condition	update action
t_0	True	x:=e
t_1	e.use_stmt=x.stmt	y:=e, y_len:=y_len+1
t_2	e.use_stmt=y.stmt	y:=e, y_len:=y_len+1
t_3	e.stmt=y.stmt	z:=e
t_{chk0}	C_{chk0}: ¬ is_alive(y.def_stmt)	-
t_{chk1}	C_{chk1}: ¬ is_alive(y.def_stmt)	-
t_{ign0}	¬ C_{chk0}	-
t_{ign1}	¬ C_{chk1}	-

Figure 4: EFSM for the event expression in Figure 3

expression shown in Figure 3. The full translation algorithm is presented in [30]. States s0-s3 and transitions t0-t3 are determined from the PATTERN clause in a way similar to that of determining FSM states from a regular expression. Transitions t_{chk0} and t_{chk1} detect the condition under which the matching cannot continue (e.g., waiting for the use of a definition that is already been killed) and trigger backtracking. Transitions t_{ign0} and t_{ign1} model non-determinism introduced by the sequential operator. Registers x, y, y_len, and z are determined from event variables. They are updated when an incoming event (represented as e) is matched. Finally, the enabling conditions of transitions are determined from the SATISFIY clause.

The time complexity of EFSM execution is $O(L \times B)$, where L is the length of event sequence and B is the number of backtracking. It can be shown that the upper bound of B is exponential to $n \times w + v - n$, where v is the total number of event variables, n is the number of event variables in Kleene closures, and w is the maximal repetition count of Kleene closures. For efficiency reason, we bounded the value of w in our experiment.

Having discussed how to specify and match context patterns, we next present our studies on the research questions.

5. Study of RQ1

In Section 3, we have outlined an approach to deriving context patterns by capturing the mechanism of how the execution of faults triggers failures. To evaluate whether this approach is feasible, we conducted a case study on common fault types.

The first step of our study is to identify a set of common fault types. To this end, we refer to a recent field study reported by Durães and Madeira [13]. They investigated 12 open source C programs ranging from user software (like vi) to system software (like RDBMS engine), and classified in total 668 real world faults using the *Orthogonal Detect Classification* (ODC). In Table 2, we summarize their results and enumerate 13 most common fault types reported in their study. These fault types cover about 90% of the real world faults reported in [13] and serve as the subjects of our case study.

Table 2: Important fault types for C programs [13][2]

ODC class	Fault type	%
Assignment (22.8%)	A1: Missing assignment	43%
	A2: Wrong/extraneous assignment	37%
	A3: Wrong assigned variable	11%
	A4: Wrong data types or conversion	7%
Check (26.6%)	C1: Missing OR-term/AND-term	47%
	C2: Wrong logic or relational operators	32%
	C3: Missing branching around statements	20%
Interface (7.8%)	I1: Wrong actual parameter expression	63%
	I2: Missing return	18%
	I3: Wrong return expression	14%
Algorithm (42.7%)	G1: Missing the whole "if" statement	40%
	G2: Missing function call	26%
	G3: Wrong function call	8%

The next step of our study is to investigate whether we can specify context patterns for these fault types based on the mechanism of how their execution triggers failures. Twelve context patterns for these 13 fault types were identified. A couple of similar fault types are covered by one pattern. Due to space limitation, we only report the results of the four patterns that involve missing statements in this paper. Readers are referred to [30] for the discussion of all the 12 context patterns.

Results on Missing Statement Faults

A1: Missing assignment. Voas and Miller [28] showed that for the omission of v=E to result in a failure, the obsolete value of v should be used after the omission place and before it is killed. Besides, the obsolete value of v shall propagate to the output. We can use the following pattern to capture this failure-triggering mechanism:

```
NAME      MISSING_ASGN
PATTERN   asgn x, any y, (use z)+, output w
SATISFY   first(z).dep_stmt == x.stmt&&
          next(z).dep_stmt == prev(z).stmt&&
          w.stmt == last(z).stmt
REFINE    line_number(y.stmt)
ATTRS     x.var_name
```

In this pattern, x matches the previous definition of v; y matches the statement preceding the omitted assignment on v; z and w match the propagation of the obsolete value.

I2: Missing return. Faults of this type result in extraneous statement executions after the position where a "return" statement is omitted. For these faults to trigger the failures, the result of these extraneous executions must affect the output. This failure-triggering mechanism is captured in the following context pattern.

```
NAME      MISSING_RET
PATTERN   entry x, any y; bran_exit z, asgn w
          (return p) & ((use q)+, output r)
SATISFY   z.func == x.func && p.func == x.func &&
```

[2] The data in Table 2 is normalized to exclude the 41 faults involving design or requirement changes.

```
         first(q).dep_stmt == w.stmt &&
         next(q).dep_stmt == prev(q).stmt &&
         r.stmt == last(q).stmt
REFINE   line_number(y.stmt)
```

In this pattern, x matches the entry of the function missing the "return" statement; y matches the statement preceding the absent "return" statement. As the absent "return" statement must be at the end of a conditional branch, y must be immediately followed by a branch exit, which is matched by z. After that, w matches the extraneous assignment before the function returns to its caller; and z and w match the propagation path of this extraneous assignment to the output.

G1: Missing "if" statement. As suggested in [24], this type of code omission is usually caused by: (1) forgetting to check buffer boundary, (2) forgetting to handle special function return code, or (3) forgetting to handle special value of the function parameter. Faults of the first reason can be located with a boundary checker and might not need CBFL. Faults of the second reason trigger a failure if the return value of the callee function is same as a certain constant value that necessitates the execution of the absent branch, and the omission of this execution affects the output. While we are still investigating techniques to capture the latter part of this mechanism, the former part can be captured in the following pattern:

```
NAME     MISSING_RET_CHECK
PATTERN  asgn x, return y, ~(use z)
SATISFY  x.asgn_type == formal_out &&
         y.func==x.func && z.dep_stmt==x.stmt
REFINE   location(y. callsite)
ATTRS    x.value
```

In this pattern, x and y match the return statement of the callee function and x.value represents the return value. The negation expression ~(use z) specifies that this return value is never used.

The context pattern for faults of the third reason is derived in a similar way. Interested readers can find it in [30].

G2: Missing function call. As observed by Dallmeier and colleagues [9], faults of this type usually trigger the failures through a sequence of function calls related to the absent function call. For example, missing the call to "open" triggers the failure only when calls to "read" or "write" are followed. Library functions are usually related by a system handle. User-defined functions, however, require static code analysis to discover their relationships. Here we show a context pattern that captures the failure-trigger mechanism of missing library function call.

```
NAME     MISSING_LIB_CALL
PATTERN  (call_lib_func x)+, any y,
         (call_lib_func z)+
SATISFY  next(x).handle == prev(x).handle &&
         first(z).handle==first(y).handle &&
         next(z).handle == prev(z).handle &&
         len(x)<=5 && len(z)<=5
REFINE   location(y.stmt)
ATTRS    x.lib_func_name, z.lib_func_name
```

In this pattern, y matches the statement preceding the absent library function call; x and z match the sequence of library function calls before and after the absent call, respectively. In the pattern, the attribute *handle* refers to the system handle used as an argument to the library function call. Besides, we follow the decision made in [9] to restrict the length of interested function call sequences via limiting the lengths of x and z.

Discussion

The results of our case study provide initial evidence that the failure-triggering mechanism of common fault types can be captured in the format of context patterns. Nevertheless, we refrain from drawing a strong conclusion because our study only involves limited types of fault. In the future, other common fault types, such as those related to concurrency, will be investigated.

As the results of our case study shows that the coverage refinement approach is applicable, we proceed to study its relevance and effectiveness.

6. Controlled experiments for RQ2 and RQ3

The goal of our experiments is to empirically investigate research questions RQ2 and RQ3 stated in Section 3:

RQ2.1: How often does coincidental correctness occur?

RQ2.2: How severely does the occurrence of coincidental correctness affect the effectiveness of CBFL?

RQ3: How effectively does coverage refinement address the coincidental correctness problem in CBFL?

In this section, we first describe the experiment setup and the threats to validity, and then present the results of the experiments. We conclude this section with our experience on real faults.

6.1 Empirical setup

Implementation

We have developed a prototype tool for coverage refinement. This tool contains three components: event instrumentation layer, EFSM executor, and CBFL technique (we chose Tarantula). The event instrumentation layer builds upon the ATAC dataflow analysis tool [4] and decomposes a test run into a sequence of events defined in Table 1. The EFSM executor matches the 12 context patterns we identified in the case study against event sequences. As shown in Section 4.3, the time complexity of pattern matching is mainly determined by the maximal repetition count of Kleene closures. For efficiency reason, we set the bound of this count as 10. In our experiments, we also used scripts from the SIR infrastructure [12] to automate test execution and code coverage collection.

All experiments reported in this section are conducted in a Sun Linux cluster with 20 nodes running CentOS 5,

Table 3: Subject programs

Program	Lines of Executable Code	Test cases	Generated Mutants	Mutants with failure	Failure rate
space	6,218	13585	40241	35008	12.4%
grep	8,164	613*8 #	52140	24588	11.7%
bc	5,381	5000	29725	15548	9.63%

#: We applied the 613 RE patterns from SIR on eight different text files.

x86_64 edition. The time overhead of event instrumentation and pattern matching is approximately 300 times to the program execution.

Subject Programs

For our study, we used three real world C programs `space`, `grep`(2.0), and `bc`(1.06) as the subjects. Table 3 provides their basic information. For each program, we list the lines of executable code (second column) and the number of available test cases (third column). `space` and `grep` were obtained from SIR [12], while `bc` was downloaded from the website of Liu [22].

These three subject programs are intended to represent three different kinds of C programs. `space` is a parser for the Array Definition Language [17]. It mainly involves operations on dynamically allocated structures. `grep` is a command line text search utility. It mainly involves operations on strings. `bc` is an interpreter for an arbitrary precision numeric processing language used in scientific computation. It mainly involves numerical computation. We believe that these three programs represent a wide range of real world C programs.

Faults Generation

To generate faults for our experiment, we first configured Proteum [10] to generate all possible program mutants with fault types listed in Table 2. We then executed these mutants using the whole test pool and excluded those without failure. Next, for each program, we randomly sampled 1000 mutants in proportion to the occurrence frequency of their fault types in Table 2. For example, according to Table 2, missing single assignment takes up 9.8% of the cases. Therefore, we randomly selected 98 mutants from those versions with such fault.

Table 3 summarizes the detail of our experiment subjects and mutants. For each program, we list the number of generated mutants (fourth column), mutants with failure (fifth column), and the average failure rate among mutants with failure (sixth column).

Metric of Fault Localization Effectiveness

CBFL techniques help programmers locate the faults by directing their attention to a set of likely faulty statements. To evaluate their effectiveness, a straightforward metric is to measure how often the set of likely faulty statements they suggest contains the real faulty statements. Following the convention in information retrieval research, we refer to this measurement as *recall*.

To measure the effectiveness of Tarantula with recall, we assume that programmers can afford to examine up to k percentage of total statements, and they follow the decreasing order of failure-correlation values when they examine the statements. Therefore, given a set of faulty programs, the recall of Tarantula is computed by:

$$recall = \frac{\begin{array}{c} number\ of\ faulty\ programs\ whose\ fault\ can \\ be\ found\ \ by\ examining\ up\ to\ k\%\ of\ the\ code \end{array}}{total\ number\ of\ \ faulty\ programs}$$

To avoid the boundary effect, we choose four different values of k as 1, 5, 10, and 15. These should be reasonable values in real world debugging scenarios. Due to space limitation, we only reported the result for k=1 and k=5, the complete result can be found in [30].

Experiment Design

To investigate RQ2.1, we instrumented mutants using gcov and executed them on all available test cases. We judged the test outcome of each test run by comparing the output with that produced by running the correct program on the same test case. Passed test runs where coincidental correctness occurs were identified by checking whether the known faulty statements have been executed.

To investigate RQ2.2, we constructed test sets with varying *concentration of coincidental correctness*, that is, the percentage of passed test cases in a test set that induce coincidental correctness. We employed 11 levels of concentration 0%, 10%, …, 100%, and thus generated 11 test sets for each mutant. Following common empirical setup for mutation analysis (e.g. [4]), we choose 100 as the size of test set. Besides, the number of failed test cases is controlled as 10, which is consistent with the average failure rate shown in Table 3. The three kinds of test cases (failed, passed with/without inducing coincidental correctness) were randomly sampled from the test pool. After test sets were constructed, we then ran Tarantula on each of them and computed the recall at each level of concentration.

To investigate RQ3, we reused all the test sets that have been constructed. We computed the recall of Tarantula in the same way as we did for RQ2, except that we applied all the 12 context patterns derived in our case study (Section 5) to produce the refined coverage for Tarantula. Note that a statement can have multiple entries in the fault localization result on refined coverage. To perform a fair comparison, for each statement, we only retain the coverage entries with the highest failure-correlation.

6.2 Threats to validity

A threat to the validity of our experiment result is that we only generated mutants with limited types of faults. To address the threat, we have covered the fault types that have been identified as the most common types in a recent field study [13]. Besides, we also cross-validated our result using real faults (see Section 6.4). Thus, the results are still of practical importance. Another possible threat is that we only used three programs. However, we have carefully selected them in order to cover different types of

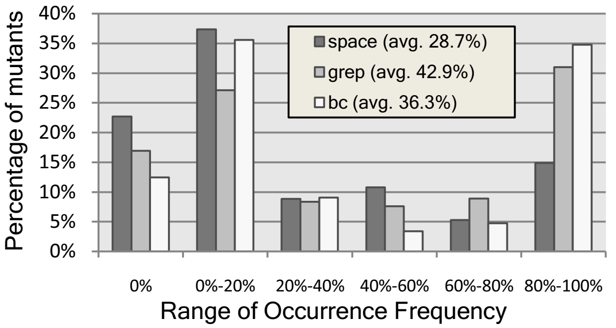


Figure 5: How often coincidental correctness occurs

programs. And these programs have been used in many studies (e.g., [8][17][18][19][22][25]). Finally, we only generate mutants with single fault. In practice, a faulty program can contain multiple faults. This issue should be addressed in future experiments.

6.3 Result and discussion

In this section, we present the experiment results and discuss how they address the three research questions.

RQ 2.1: The Frequency of Coincidental Correctness

Figure 5 shows the frequency distribution of coincidental correctness for each program. The horizontal axis shows the range of occurrence frequency of coincidental correctness, and the vertical axis shows the percentage of mutants whose occurrence frequency of coincidental correctness falls into each range. For example, for grep, there are 31% of the mutants for which coincidental correctness occurs in 80%-100% of the passed test runs.

By inspecting Figure 5, we can observe that coincidental correctness is common. Take space for example, on average it occurs in 28.6% of the passed test runs. Besides, there are a considerable number of mutants (15% for space, 31% for grep, 35% for bc) for which coincidental correctness occurs very frequently (over 80%).

RQ 2.2: The Impact of Coincidental Correctness

Figure 6 illustrates the impact of coincidental correctness on Tarantula when statement coverage is used for computing the failure-correlation value. The horizontal axis shows the concentration of coincidental correctness in the test set, and the vertical axis shows the recall, that is, the percentage of mutants whose fault is among the top *k*% likely faulty statements ranked by Tarantula using the failure-correlation values. For each program, we conducted the experiment twice with the value of *k* being 1 and 5, respectively. The data corresponds to the gray curves marked with boxes in Figure 6. As code omission faults are of special interest, we also show the result for this sub-category in Figure 7.

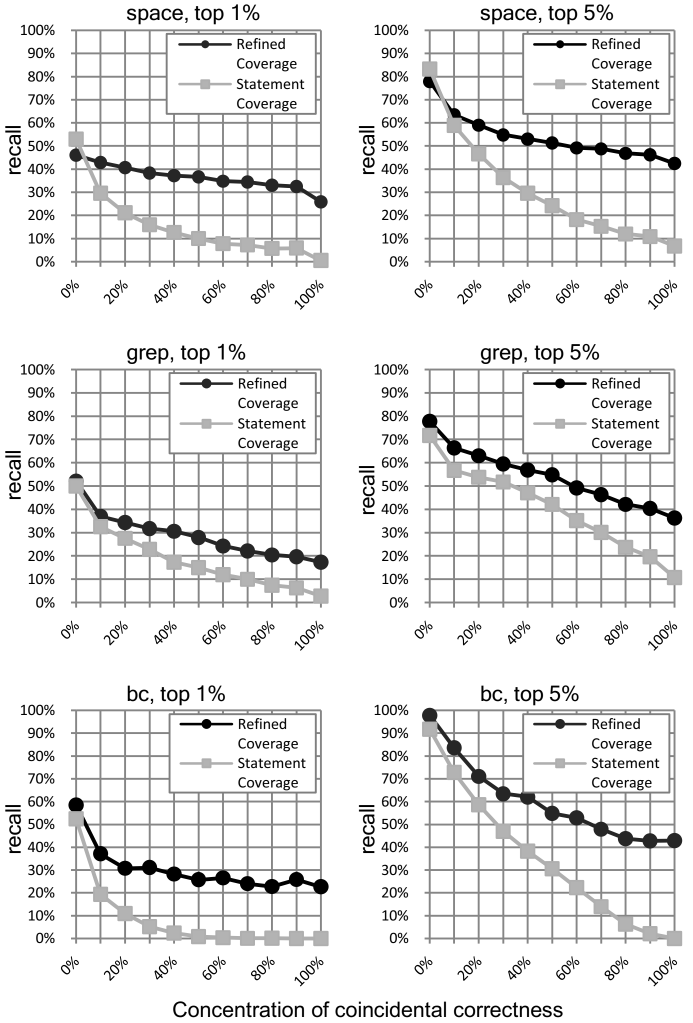


Figure 6: RQ2.2 and RQ3, Overall Results

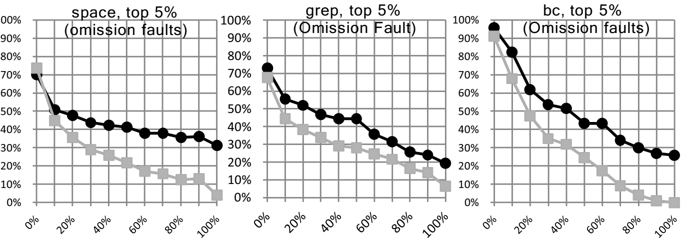


Figure 7: RQ2.2 and RQ3, Code Omission Faults

The result of the experiment shows that coincidental correctness can be harmful to CBFL. From Figure 6, we can observe that the effectiveness of Tarantula declines significantly when the occurrences of coincidental correctness increase. This is consistent with the observation made by Jones and colleagues [18] that CBFL has difficulty locating faults in the main program path and initialization code — in fact, we found these faults induce the most occurrences of coincidental correctness.

RQ3: The Effectiveness of Coverage Refinement

Let us examine the third research question. To investigate how effectively coverage refinement alleviates the problem of coincidental correctness on CBFL, we have reused the test sets constructed for RQ2, and applied the refined coverage rather than the original statement cover-

Table 4: Results with real faults in space

Faulty versions	%COR(in test pool)	Statement Coverage		Refined Coverage	
		failure correlation	higher (equal)	failure correlation	higher (equal)
v12	0.15%	1.000	0 (17.0)	1.000	0 (82.1)
v18	0.15%	1.000	0 (45.3)	1.000	0 (55.6)
v15	55.09%	0.646	172.3 (25.3)	0.982	30.1 (15.9)
v14	72.23%	0.605	186.3 (17.0)	0.953	28.5 (11.3)
v3	95.04%	0.512	456.6 (57.4)	0.609	350.3(35.6)
v29	99.77%	0.503	905.6 (309.7)	1.000	0 (23.0)

age to Tarantula for locating faults. The data corresponds to the black curves that are marked with circles in Figure 6 and Figure 7.

From Figure 6, we observe that coverage refinement considerably improves the effectiveness of CBFL. In the top-5% case, even when coincidental correctness occurs in all the passed test runs, Tarantula can still locate the fault in 35-42% of the studied cases if refined coverage is used. By contrast, using the original statement coverage, Tarantula mostly fails to locate the fault in such situation.

Let us look closer at the data. One observation is, although the improvement in the case of grep is still significant, it becomes only noticeable in the case of space or bc. We conjectured that this is due to the bound of Kleene closure in our implementation of EFSM executor. As grep handles strings, the data propagation path in it is considerably longer than that in space or bc. Therefore, the bound on the repetition count of Kleene closure might result in inaccuracy. In the future, we should conduct studies to investigate this trade-off. Another observation is that when coincidental correctness occurs in few passed test runs (e.g., < 20% of the total passed test runs), the improvement of coverage refinement on Tarantula, if any, is marginal. However, from Figure 5, we can observe that for around half of the faults, coincidental correctness occurs in more than 20% of passed test runs. For these faults, coverage refinement is useful.

6.4 Experiences on real faults

Having conducted the experiments with mutants, we further validated the results using 38 faulty versions of space (also obtained from SIR) containing real faults. For each of them, we randomly sampled ten failure-revealing test sets of size 100 from the test pool. These test sets are intended to simulate the random test sets that programmers use for debugging in the real world. Using these test sets, we compared the performance of Tarantula on statement coverage and refined coverage.

Table 4 reports the result for faulty versions with the lowest, medium, and the highest likelihood of coincidental correctness. In this table, the column "%COR" presents the percentage of coincidentally passed test cases in the test pool. The column "failure correlation" presents the average failure-correlation value of the faulty statement. The column "higher (equal)" presents the average number of non-faulty statements whose failure-correlation value is higher than or equal to that of the faulty statement.

From the result, we observe that coverage refinement significantly improves the performance of Tarantula when coincidental correctness frequently occurs. As a representative example, the following faulty version of space shows how this improvement is achieved.

Version 14:

```
 int unifamp(…){
8801:  error=0;
       /*Missing: error= */
8805:  GetKeyword(Keywords[88], curr_ptr);
8807:  if (error != 0) { …… };
```

The fault in this case is equivalent to missing the "if" statement at line 8807, which triggers the failure only when the return value of GetKeyword is one (indicating that the keyword is found). The context pattern MISSING_RET_CHECK shown in Section 4.2 captures this failure-triggering mechanism and strengthens the correlation between program failures and the coverage of faulty statement (line 8805). This results in significant reduction in effort to examine the code before locating this fault (186.3 vs. 28.5) by a developer.

7. Conclusion

Coverage-based fault localization (CBFL) techniques exploit the correlation between program failures and the coverage of faulty statements to locate faults. The occurrence of coincidental correctness can adversely affect their effectiveness. In this paper, we have proposed an approach to address this problem. We observe when the execution of faults triggers failures; some control flow and data flow patterns would appear. Such patterns can be derived empirically or analytically and used across different programs. With these patterns, we refine code coverage to strengthen the correlation between program failures and the coverage of faulty statements, making it easier for CBFL techniques to locate faults. We have evaluated the feasibility and effectiveness of our approach through empirical investigations. The results show that coverage refinement with context patterns is a promising approach to address the coincidental correctness problem in CBFL.

8. References

[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*, Pages 89-98, Sep., 2007.

[2] H. Agrawal and J.R. Horgan, Dynamic program slicing, In *Proc. of PLDI'90* , Pages 246 - 256, Nov., 1990.

[3] H. Agrawal, J. Horgan, S., Lodon, and W. Wong, Fault Localization using Execution Slices and Dataflow Tests, In *Proc. of ISSRE'95*, Pages 143-151, Oct., 1995.

[4] J.H. Andrews, L.C. Briand, and Y. Labiche, Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria, *IEEE TSE*, 32(8):608-624, 2006.

[5] B. Baudry, F. Fleurey, and Y. Le Traon, Improving test suites for efficient fault localization, In *Proc.of ICSE'06*, Pages 82 - 91, May, 2006.

[6] P.C. Bates, Debugging Heterogeneous Distributed Systems using Event-based Models of Behavior, *ACM Transactions on Computer Systems*, 13(1): 1-31,1995.

[7] K.T. Cheng, and A.S. Krishnakumar, Automatic funct-ional test generation using the extended finite state machine model, In *Proc. of DAC'93*, Pages 86-91, 1993.

[8] H. Cleve, and A. Zeller, Locating Causes of Program Failures, In *Proc. of ICSE'05*,Pages: 342 -351, May, 2005.

[9] V. Dallmeier, C. Lindig, and A. Zeller, Lightweight Detect Localization for Java, In *Proc. of ECOOP'05*, Pages 528-550, Jul., 2005.

[10] M.E. Delamaro, and J.C. Maldonado, Proteum-A Tool for the Assessment of Test Adequacy for C Programs, In *Proc. of PCS'96*, Pages 79-95, Jul., 1996.

[11] M. Daran, Software Error Analysis: A Real Case Study Involving Real Faults and Mutations, In *Proc. Of ISSTA'96,* Pages: 158-171, Oct., 1996.

[12] H.S. Do, S.G. Elbaum, and G. Rothermel, Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact, *Empirical . Softw. Eng.*, 10(4):405-435, Oct., 2005.

[13] J.A. Durães, and H.S. Madeira, Emulation of Software Faults: A Field Data Study and a Practical Approach, *IEEE TSE*, 32(11):849-867, Nov., 2006.

[14] I. Forgács and A. Bertolino, Preventing untestedness in data-flow testing, *Softw. Test., Verif. Reliab.* 12(1):29–58, 2002.

[15] R.M. Hierons, Avoiding coincidental correctness in boundary value analysis, *ACM Trans. Softw. Eng. Methodol.*, 15(3): 227-241, Jul., 2006.

[16] J.R. Horgan, and S. London, Data flow coverage and the C Language, In *Proc. of ISSTA'91*, Pages 87-97, Oct., 1991.

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, Experiments of the Effectiveness of Dataflow and Controlflow Based Test Adequacy Criteria, In *Proc. of ICSE'94*, Pages 191 - 200, May, 1994.

[18] J.A. Jones, M.J. Harrold, and J. Stasko, Fault Localization Using Visualization of Test Information, In *Proc. of ICSE'02*, Pages 54-56, May, 2002.

[19] J.A. Jones, and M.J. Harrold, Empircal Evaluation of the Tarantula Automatic Fault-Localization Technique, In *Proc. of ASE'05*, Pages: 273-282, Nov., 2005.

[20] S.H. Kim, K. Pan, and E.E.J. Whitehead, Memories of Bug Fixes, In *Proc. of FSE'06*, Pages: 35-45, Nov., 2006.

[21] B. Liblit, A. Aiken, A.X. Zheng, and M. I. Jordan, Bug isolation via remote program sampling, In *Proc. of PLDI'03*, Pages 141-154, Jun., 2003.

[22] C. Liu, L. Fei, X.F. Yan, J.W. Han, and S. Midkiff, Statistical Debugging: a Hypothesis Testing-Based Approach, *IEEE TSE*, 32(10):1-17, 2006.

[23] B. Marick, The Weak Mutation Hypothesis, In *Proc. of ISSTA'91*, Page 190-199, Oct., 1991.

[24] B. Marick, Faults of Omission, *Software Testing and Quality Engineering Magazine*, 2(1), 2000.

[25] M. Renieris, and S. Reiss, Fault Localization With Nearest Neighbor Queries, In *Proc. of ASE'03*, Pages 30-39, Oct., 2003.

[26] D, J. Richardson, and M.C. Thompson, An Analysis of Test Selection Criterria Using the RELAY Model of Fault Detection, *IEEE TSE*,19(60):533-553, 1993.

[27] I. Vessey, Expertise in Debugging Compute Programs, *Inter. J. of Man-Machine Studies*, 23(5):459-494, 1985.

[28] J.M. Voas and K.W. Miller, Applying a Dynamic Testability Technique to Debugging Certain Classes of Software Faults, *Software Quality Journal*, 2:61-75, 1993.

[29] Y.B. Yu, J.A. Jones, and M.J. Harrold, An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization, In *Proc. of ICSE'08*, Pages 201-210, 2008.

[30] X.M. Wang, S.C., Cheung, W.K. Chan, and Z.Y. Zhang, Taming Coincidental Correctness: Refine Code Coverage with Context Pattern to Improve Fault Localization, HKUST-CS08-05, 2008.

[31] E. Wong, Y. Qi, L. Zhao, and K.Y. Cai, Effective Fault Localization using Code Coverage, In *Proc. of COMPSAC'07*, Pages 449-456, Jul., 2007.

[32] E. Wong, and Y. Qi, Effective program debugging based on execution slices and inter-block data dependency, *Journal of Systems and Software*, 79(2):891–903, 2006.

[33] X.Y. Zhang, S. Tallam, N. Gupta, R. Gupta, Towards locating execution omission errors, In *Proc of PLDI'07*, Jun., Pages: 415-424, 2007.

[34] A. Zeller, Isolating cause-effect chains from computer programs, In *Proc. of FSE'02*, Pages: 1-10, Nov., 2002.

Appendix A: Semantics of event expression

Let $e[1,...,n]$ be a sequence of events e_1, e_2 …, e_n. The semantics of event expression is described as follows:

Base case: The expression "$T\ x$" matches $e[1,...,n]$ if and only if n=1 and the type of e_1 is T.

Sequential: "$\mathtt{E}_1\mathtt{,}\ \mathtt{E}_2$" matches $e[1,...,n]$ if and only if there exist two integers i and j ($1 \le i < j \le n$) such that $\mathtt{E}_1$ matches $e[1,...,i]$ and $\mathtt{E}_2$ matches $e[j,...,n]$. "$\mathtt{E}_1\mathtt{;}\mathtt{E}_2$" matches $e[1,...,n]$ if and only if there exist an integer i ($1 \le i < n$) such that $\mathtt{E}_1$ matches $e[1,...,i]$ and $\mathtt{E}_2$ matches $e[i+1,...,n]$.

Kleene Closure: "$\mathtt{E+}$" matches $e[1,...,n]$ if and only if there are k ($k \ge 1$) subsequences $e[a_1,..,a_2]$, $e[a_3,..,a_4]$…, $e[a_{2k+1},.., a_{2k+2}]$ ($1=a_1 \le a_2 < a_3 \le a_4 \ldots \le a_{2k+1} \le n$) of $e[1,...,n]$ such that $\mathtt{E}$ matches each of them.

Concurrency: "$\mathtt{E}_1\ \mathtt{\&}\ \mathtt{E}_2$" matches $e[1,...,n]$ if and only if there exists an integer i ($1 \le i \le n$) such that either $\mathtt{E}_1$ matches $e[1,...,n]$ and $\mathtt{E}_2$ match $e[i,...,n]$, or $\mathtt{E}_1$ matches $e[i,...,n]$ and $\mathtt{E}_2$ match $e[1,...,n]$.

Choice: "$\mathtt{E}_1|\mathtt{E}_2$" matches $e[1,...,n]$ if and only $\mathtt{E}_1$ matches $e[1,...,n]$ or $\mathtt{E}_2$ matches $e[1,...,n]$.

Negation: "$\mathtt{E}_1\mathtt{,\sim E,}\mathtt{E}_2$" matches $e[1,...,n]$ if and only if there exist two integers i and j ($1 \le i < j \le n$) such that $\mathtt{E}_1$ matches $e[1,...,i]$, $\mathtt{E}_2$ matches $e[j,...,n]$, and $\mathtt{E}$ does not match e$[i+1,...,j-1]$. "$\mathtt{E}_1\mathtt{,\sim E}$" and "$\mathtt{\sim E,}\mathtt{E}_1$" are similarly defined.