Statistical Debugging with Elastic Predicates

Ross Gore, Paul F. Reynolds, Jr., David Kamensky
Dept of Computer Science
University of Virginia
Charlottesville, VA, 22901
{rjg7v, pfr, dmk3d}@virginia.edu

Abstract— Traditional debugging and fault localization methods have addressed localization of sources of faults while assuming discrete data types. While these methods are effective in general, they are not tailored to an important class of software, including simulations and computational models, which employ floating-point computations and continuous stochastic distributions to represent, or support evaluation of, an underlying model. To address this shortcoming, we introduce elastic predicates, a novel approach to predicate-based statistical debugging. Elastic predicates are based on values profiled at instrumented points in a program, creating partitions of test cases that more closely match the subspaces where a fault is and is not expressed. We present experimental results for established fault localization benchmarks and widely used simulations that show improved effectiveness with elastic predicates over existing fault localization techniques. These improvements persist in the face of sparse test suites and sparse instrumentation sampling rates.

Keywords- automated debugging; fault localization

I. INTRODUCTION

Our interest is in exploratory software. Exploratory software comprises simulations, computational models and other software deployed to gather insight into uncertainties in an underlying model. These uncertainties are often reflected through stochastic distributions, and floating-point computations that generally accompany them. Exploratory software has become a common tool for subject matter experts (SMEs) in disciplines from the physical sciences to economics [1, 2]. Predictions based on exploratory software outcomes have entered the mainstream of critical public policy and research decision-making practices, often affecting large numbers of people and valuable resources.

Unfortunately SMEs can struggle with the resolution of unexpected exploratory software outcomes. Unexpected outcomes can reflect new knowledge about the underlying model, or a fault. Currently there is no known automated analysis method for separating them. These challenges represent the basis of our motivation. Separating unexpected valid exploratory software outcomes from failures is an interesting, difficult problem. We have not solved it. However, our predicate-based statistical debugger, Exploratory Software Predictor (ESP), does localize sources of unexpected outcomes effectively in established fault localization benchmarks and widely used simulations. ESP is novel, and it extends the domain of programs for which fault localization analysis has been shown to be effective.

ESP is a predicate-based statistical debugging approach focused on identifying single or multiple sources of unexpected outcomes in exploratory software. Predicate-based statistical debugging approaches, such as ESP, represent a class of fault localization techniques that share a common structure. Each approach consists of a set of conditional propositions, or predicates, tested at particular program points. The predicates are given an importance score based on how frequently they are true in the passing and failing test cases for a failing program. A single predicate can be thought of as partitioning the space of all test cases into two subspaces: those satisfying the predicate and those not. Better predicates create partitions that more closely match the subspaces where the fault is and is not expressed [3, 4]. The predicates are ranked, based on their importance score. Rankings and scores are provided to SMEs to assist in finding and fixing faults.

Existing predicate-based statistical debugging approaches do not employ predicates for floating-point computations because of the additional amount of space and time floating-point variables require compared to character, Boolean, integer and pointer types. Furthermore, the predicates employed are uniform and static. When applying predicate-based statistical debugging to exploratory software these design decisions present opportunities for improvement.

Predicates in existing techniques are uniform in the sense that the same set of conditional propositions is tested at each program point. The predicates are static because each conditional proposition being tested is determined before the execution of the program. While the existing techniques employing static and uniform predicates have been shown to be effective for a variety of general-purpose software applications, they can be improved. An example elucidates the improvements that are possible and begins to show how these improvements are achieved in ESP.

Fig. 1 shows the source code of the more_arrays() function in the 1.06 version of the GNU implementation of BC, a basic command-line calculator tool [5]. The more_arrays() function is responsible for increasing the
number of arrays needed for computing. The logic within the function is an example of buffer reallocation. Line 167 allocates a larger chunk of memory. Line 171 is the top of a loop that copies values over from the old, smaller array. Line 176 completes the resize by zeroing out the new extra space. However, there is a fault in the function. The allocation on line 167 requests space for a_count items. The copying loop on line 171 ranges from 1 through old_count - 1. The zeroing loop on line 176 continues on from old_count through v_count - 1. Here lies the fault: the new storage buffer has room for a_count elements, but the second loop is incorrectly bound by v_count. Thus, when v_count is larger than a_count the program fails.

In the canonical predicate-based statistical debugger Cooperative Bug Isolation (CBI), three static predicates are tested for each assignment statement to each variable x in BC. These predicates are: \((x>0), (x=0)\) and \((x<0)\). Unfortunately, these predicates do not make the fault within more_arrays() easy to discern. The predicates are chosen before the program is executed and are the same at each instrumented program point. Thus, they are unable to adapt when most of the values assigned to the variables in more_arrays() are greater than zero, and satisfy the same predicate, \((x>0)\). The result is predicates that do not closely match the subspaces where the fault is and is not expressed.

Elastic predicates address this problem. For each program point, similar values are clustered together to form a predicate. In the example, an elastic predicate clusters together unusually large values for indx in line 176 and captures the fault. The predicate suggests that failures frequently occur when the input to BC defines an unusually large number of arrays. The location of the fault and the cause of the failure are clear after identification and explanation. However, this fault was present and undiscovered for ten years in BC [3].

The elastic predicate is an improvement over the existing static and uniform predicates because it profiles the instrumented program points during execution to identify partitions of test cases that more closely match the subspaces where an unexpected outcome is and is not expressed. The result is improved effectiveness. Based on our experimental evaluation, the most dramatic improvements occur when the cause of an unexpected outcome lies in a statement containing floating-point computations.

II. ELASTIC PREDICATES

A. Importance Scores

The extent to which a predicate matches these subspaces is measured through an importance score. Both elastic predicates and uniform and static predicates require two data structures for each executed test case to compute importance scores. The two data structures are: a one bit feedback report, R, indicating if the test case was passed or failed and a vector V with an entry for each predicate. Within V each entry indicates the number of times the corresponding predicate is observed to be true during test case execution. Once all test cases have been executed, the importance scores for each predicate can be calculated [3, 4].

Several different formulas to compute importance scores for predicates have been explored. Research showed that predicates that are sensitive and specific should yield high importance scores. Sensitivity predicates account for a high percentage of failed test cases and specific predicates do not predict failure for successful test cases. The data for predicate \(p\) from each feedback report \(R\) and each corresponding \(V\) is aggregated into four measures [4]:

1. \(S(p\ \text{obs})\) and \(F(p\ \text{obs})\) - the number of successful and failed test cases in which \(p\) was evaluated.
2. \(S(p)\) and \(F(p)\) - the number of successful and failed test cases in which the value of \(p\) was found to be true.

Sensitivity: \(\log(F(p)/\log(\text{MaxF})\). \text{MaxF}\) is the maximum times the predicate \(p\) is satisfied in failing runs. This is the percentage of the failing test cases the predicate accounts for.

Specificity: \(\text{Increase}(p)\) is the amount \(p\) being true increases the probability of failure over reaching where \(p\) is defined.

\[
\text{Increase}(p) = \frac{F(p)}{S(p) + F(p)} - \frac{F(p \text{obs})}{S(p \text{obs}) + F(p \text{obs})}
\]

(1)

Sensitivity and specificity are combined via their harmonic mean. This metric is the importance score for the predicate.

\[
\text{Importance}(p) = \frac{1}{\text{Increase}(p)} + \frac{1}{\log(F(p)/\log(\text{MaxF}))}
\]

(2)

Generating an optimal elastic predicate for a program point requires all program point values to be stored and sorted, and the set of contiguous values that maximize the

<table>
<thead>
<tr>
<th>void</th>
<th>more_arrays()</th>
</tr>
</thead>
<tbody>
<tr>
<td>152</td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>{</td>
</tr>
<tr>
<td>154</td>
<td>int indx;</td>
</tr>
<tr>
<td>155</td>
<td>int old_count;</td>
</tr>
<tr>
<td>156</td>
<td>bc_var_array **old_ary;</td>
</tr>
<tr>
<td>157</td>
<td>char *old_names;</td>
</tr>
<tr>
<td>158</td>
<td>}</td>
</tr>
<tr>
<td>159</td>
<td>}</td>
</tr>
<tr>
<td>160</td>
<td>}</td>
</tr>
<tr>
<td>161</td>
<td>}</td>
</tr>
<tr>
<td>162</td>
<td>}</td>
</tr>
<tr>
<td>163</td>
<td>}</td>
</tr>
<tr>
<td>164</td>
<td>}</td>
</tr>
<tr>
<td>165</td>
<td>}</td>
</tr>
<tr>
<td>166</td>
<td>}</td>
</tr>
<tr>
<td>167</td>
<td>}</td>
</tr>
<tr>
<td>168</td>
<td>}</td>
</tr>
<tr>
<td>169</td>
<td>}</td>
</tr>
<tr>
<td>170</td>
<td>}</td>
</tr>
<tr>
<td>171</td>
<td>}</td>
</tr>
<tr>
<td>172</td>
<td>}</td>
</tr>
<tr>
<td>173</td>
<td>}</td>
</tr>
<tr>
<td>174</td>
<td>}</td>
</tr>
<tr>
<td>175</td>
<td>}</td>
</tr>
<tr>
<td>176</td>
<td>}</td>
</tr>
<tr>
<td>177</td>
<td>}</td>
</tr>
<tr>
<td>178</td>
<td>}</td>
</tr>
<tr>
<td>179</td>
<td>}</td>
</tr>
<tr>
<td>180</td>
<td>}</td>
</tr>
<tr>
<td>181</td>
<td>}</td>
</tr>
<tr>
<td>182</td>
<td>}</td>
</tr>
<tr>
<td>183</td>
<td>}</td>
</tr>
<tr>
<td>184</td>
<td>}</td>
</tr>
<tr>
<td>185</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 1. The source code of the more_arrays() function in BC.
importance score to be identified. Such an approach maximizes effectiveness but is impractical in practice. The space required to store each value for each program point can exceed available space and sorting and searching all the values cannot be performed fast enough to be useful. However, elastic predicates that can be computed efficiently are realizable. While these elastic predicates do not maximize the importance score at program points, our results show they approach the improvements in effectiveness that are possible with optimal elastic predicates.

B. Predicates within ESP

Numerous statistical debugging approaches have been proposed to identify predicates that are good failure predictors [3,4, 6-9]. The most effective of these approaches analyze variable values within a program. However, accounting for all possible values at each assignment to a given variable is impractical. Instead existing predicate-based statistical debuggers utilize an instrumentation scheme with uniform and static predicates at each variable assignment statement. This instrumentation scheme is referred to as single variable and is an extension of the returns scheme in CBI. The most common single variable instrumentation scheme for an assignment to, or the return of, a variable \(x\) employs these static and uniform predicates: \((x>0), (x=0)\) and \((x<0)\).

ESP complements these static predicates with elastic predicates. Computing elastic predicates in ESP is a two-step process. In the first step, ESP instruments the faulty program to accumulate the mean and standard deviation at each program point. As ESP executes test cases, the mean, \(\mu_x\), and standard deviation, \(\sigma_x\), for an assignment to, or return of, a variable \(x\) are computed using online algorithms requiring constant space [10]. Once all test cases are executed, each \(\mu_x\) and \(\sigma_x\) is stored. Then, each \(\mu_x\) and \(\sigma_x\) is used to generate these predicates:

1. \(x < (\mu_x - 3\sigma_x)\)
2. \((\mu_x - 3\sigma_x) \leq x < (\mu_x - 2\sigma_x)\)
3. \((\mu_x - 2\sigma_x) \leq x < (\mu_x - \sigma_x)\)
4. \((\mu_x - \sigma_x) \leq x < \mu_x\)
5. \(\mu_x = x\)
6. \(\mu_x < x \leq (\mu_x + \sigma_x)\)
7. \((\mu_x + \sigma_x) < x \leq (\mu_x + 2\sigma_x)\)
8. \((\mu_x + 2\sigma_x) < x \leq (\mu_x + 3\sigma_x)\)
9. \(x \geq (\mu_x + 3\sigma_x)\)

In the second step the defined elastic predicates and the static and uniform predicates are scored for importance in the same manner as the static and uniform predicates in CBI.

While the formula to compute importance scores for static and uniform predicates and elastic predicates is the same, the scores are not. The BC example highlights these differences. We use 1,000 randomly generated, valid BC programs with various sizes and complexities as test cases. This approach to generating test cases is modeled after Liu et al.’s case study of BC [6]. The top two ranked predicates for the ESP and CBI methods are shown in Table 1 and 2.

<table>
<thead>
<tr>
<th>Table I. The Top Ranked CBI Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>_filename</td>
</tr>
<tr>
<td>storage.c</td>
</tr>
<tr>
<td>storage.c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table II. The Top Ranked ESP Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
</tr>
<tr>
<td>storage.c</td>
</tr>
<tr>
<td>storage.c</td>
</tr>
</tbody>
</table>

Recall, line 176 contains the fault: the new storage buffer has room for \(a\)-count elements, but the second loop is incorrectly bounded by \(v\)-count instead. Thus, when \(v\)-count and \(indx\) are larger than \(a\)-count the program fails. The advantage of elastic predicates is clear in Table 2.

C. Scalar Pairs

Multiple variables within a program can have important relationships that cannot be captured with a single variable instrumentation scheme. The Daikon project identifies implicit invariants to aid program evolution and understanding [11]. ESP and existing predicate-based statistical debugging approaches identify near-invariants that are only violated when the program fails test cases through an instrumentation scheme called scalar pairs [3]. Within CBI the scalar pairs scheme instruments assignments to Boolean, character, integer and pointer typed variables. ESP extends the scheme to floating-point typed variables. The name, scalar pairs, refers to the data type of the variables.

In existing predicate-based approaches, the scalar pairs scheme examines possible invariants from a uniform and static set. At each assignment to, or the return of, a variable \(x\) these approaches identify all other same-typed local or global variables \(y_1, y_2, ..., y_n\) that are currently in scope. For each pair of variables the scheme compares the new value of \(x\) with the existing value of \(y_i\): \((x > y_i), (x = y_i)\) and \((x < y_i)\).

In ESP, instrumentation computes and stores mean \(\mu_{x-y_i}\) and standard deviation \(\sigma_{x-y_i}\) of the difference between the new value of \(x\) and the existing value of \(y_i\) as the test cases are executed. Then using the computed \(\mu_{x-y_i}\) and \(\sigma_{x-y_i}\), ESP complements the three static and uniform predicates used in existing techniques with nine elastic predicates which partition the values for an instrumented program point for three standard deviations above and below the mean of \(x - y_i\). These predicates are similar to the elastic predicates already presented and derived by replacing \(\mu_x\) and \(\sigma_x\) with \(\mu_{x-y_i}\) and \(\sigma_{x-y_i}\).

The possibility of improved effectiveness due to elastic predicates in the single variable or scalar pairs scheme is not without cost. ESP elastic predicates require the mean and standard deviation of each program point to be computed, which entails executing the test suite twice. The additional
execution time can offer improvements in effectiveness. ESP efficiency and effectiveness are evaluated in the next section.

### III. Evaluation

#### A. Experimental Setup

The utility of a fault localization technique is determined through experimental evaluation. Following previous work in the fault localization community, we employ the Siemens Benchmark Suite as a portion of our evaluation of ESP. The subset of benchmarks from the Siemens Suite used for our experiments is listed in Table 3. The programs, along with their corresponding faulty versions and test cases, were obtained from [12]. All Siemens faulty versions contain seeded faults. These faults are computation-related as opposed to memory-related. Most faulty versions are seeded with a single fault in a single statement, but some faulty versions involve several statements. Several faulty versions were excluded because they did not yield any failing test cases from the provided set of test cases. These versions have been excluded in previous fault localization evaluations [13, 14]. The programs in Table 3 were chosen from the Siemens Suite because (1) they contain a large amount of floating-point type computations or (2) they are a simulation. These characteristics make them good candidates to evaluate ESP against alternative approaches.

Our subset of the Siemens Suite consists of four benchmarks: tcas, totinfo, sched and sched2. tcas is an air traffic collision avoidance simulation that contains no loops and represents one conditional check spread across several functions; it takes as input a set of integer parameters and reports one of three output values. totinfo reads a collection of numeric data tables as input and computes statistics for each table as well as across all tables. Programs sched and sched2 simulate priority schedulers for processes, taking as input a number of processes and a list of commands, and outputting the processes as they complete.

We implemented ranking approaches Interesting Value Map Pairs (IVMP), ESP, OPT, CBI and Tarantula.

**IVMP.** Within IVMP, each statement's ranking is based on the number of failing executions in which a state alteration within the statement results in a passing execution [13, 14]. Any ties are broken using the Tarantula's suspiciousness formula described later in this section.

**ESP.** ESP is a predicate-based statistical debugging approach that complements static and uniform predicates with elastic predicates. The evaluation considers both the single variable and scalar pairs instrumentation schemes for ESP. Given a list of predicates ranked by importance scores statements are ranked according to the following:

1. For each statement identify the corresponding predicate with the highest importance score and move the statement and importance score to set ST.
2. Rank the statements in ST by importance score.

**OPT.** OPT generates elastic predicates yielding the maximum importance score at each program point. While OPT is infeasible in practice it serves as the upper bound of improvements from a predicate-based approach employing importance scores. It uses the ESP ranking process.

**CBI.** In our evaluation both CBI's single variable and scalar pairs instrumentation schemes are considered. Existing implementations of CBI only consider Boolean, character, integer, or pointer typed variables [3, 4]. To ensure a fair evaluation we extend CBI to consider floating-point type variables. CBI uses the ESP ranking process.

**Tarantula.** Tarantula ranks statements in order of suspiciousness \( \text{susp} \). The \( \text{susp} \) of a statement \( s \) is [15]:

\[
\text{susp}(s) = \frac{\text{failed}(s)/\text{totalFailed}}{\text{totalFailed} - \text{totalPassed}} + \frac{\text{passed}(s)/\text{totalPassed}}{\text{totalFailed} - \text{totalPassed}}. \tag{3}
\]

Here, \( \text{failed}(s) \) and \( \text{passed}(s) \) represent the number of failing and passing executions including \( s \). \( \text{totalFailed} \) and \( \text{totalPassed} \) are the total number of respective executions.

In our evaluation, we rank only those program statements that are executed by failing test cases using the test suite associated with each faulty version of each program. When multiple statements are tied for a particular rank, all tied statements are given a rank value equal to the maximum rank value from among the tied statements. This reflects the conservative assumption that a SME will examine all tied statements before any faulty statement.

To evaluate each approach we assign a score to each ranked set of statements that is the percentage of program statements executed by failing test cases in the test suite that need not be examined given the rank order of the statements. Given a ranked list of statements \( S \), where the faulty statement occurs at rank \( r \) and \( n \) total statements are executed by failing test cases the score is: 

\[
\text{score}(S) = (n-r)/n * 100.
\]

Finally, there are two details pertaining to certain types of faults. First, for versions with a faulty constant assignment statement (15 of 82 versions), we consider the statement to be examined by a SME when it directly examined or when a statement explicitly using the constant is examined. Second, for versions where the fault is a missing statement (16 of 82 versions), statements directly adjacent to the missing code qualify as the missing statement. These issues are handled the same way in previous evaluations [13, 14].

#### B. Statement Ranking Effectiveness

Our experimental results for the benchmarks in the Siemens Suite are shown for each of the statement ranking approaches in Fig. 2 and 3. In the figures the x-axis represents the lower bound of each score range, and the y-axis represents the percentage of faulty versions with a score greater than or equal to the lower bound. Fig. 2 shows the results for ESP and CBI under the single variable

<table>
<thead>
<tr>
<th>Name</th>
<th>Lines of Code</th>
<th>Versions</th>
<th># Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>138</td>
<td>41</td>
<td>1608</td>
</tr>
<tr>
<td>totinfo</td>
<td>396</td>
<td>23</td>
<td>1052</td>
</tr>
<tr>
<td>sched</td>
<td>299</td>
<td>9</td>
<td>2650</td>
</tr>
<tr>
<td>sched2</td>
<td>297</td>
<td>9</td>
<td>2710</td>
</tr>
</tbody>
</table>

---

**TABLE III. Siemens Suite Evaluation Programs**
instrumentation scheme. Fig. 3 shows the results with the addition of the scalar pairs scheme to ESP and CBI.

These presentations of data follow the convention of Jones et al. [15]. However, whereas Jones et al. computes scores with respect to the total number of program statements, we compute scores with respect to the total number of statements executed by failing test cases in the suite. Fig. 2 and 3 show that for our subset of the Siemens Suite the IVMP approach, OPT and ESP perform much better than Tarantula or CBI overall.

**ESP vs. IVMP.** Within our subset of the Siemens Suite IVMP performs better than ESP for tcas, sched and sched2. In these programs IVMP had a score of 90% or higher 40 times while ESP only had a score of 90% or higher 32 times. However, ESP performed well for the totinfo program, which frequently employs floating-point computations, and IVMP did not. Within totinfo it is very difficult for IVMP to perform state alterations that cause a failing test case to pass [13]. This difficulty is due to the level of precision in the floating-point computations that generate the program’s output. As a result the approach echoes the ranking system in Tarantula for most statements. In the evaluation IVMP performs the same as or worse than Tarantula for 15 of the 23 faulty versions of totinfo [13].

**ESP vs. CBI.** ESP performs better than CBI under both instrumentation schemes for our subset of the Siemens Suite. Under the single variable instrumentation scheme there are only 6 out of the total 82 faulty program versions where ESP assigned a lower rank to the statement containing the fault than CBI. These instances are a result of scenarios where ESP finds a number of predicates at different program points with higher importance scores than the corresponding CBI predicate, which identifies the fault. We view these versions as outlying data points in our evaluation. For the vast majority of the faulty versions, identifying predicates with high importance scores leads to the effective identification of faults. Furthermore, existing studies show that when SMEs or developers are provided with a good failure predicting predicate they are able to quickly identify the fault within a program whether or not the predicate represents the statement containing the fault [3].

The lack of significant additional effectiveness provided by the inclusion of the scalar pairs scheme for ESP and CBI is noteworthy. There are some small improvements in ESP but none in CBI. Liu et al. observed similar results for CBI with the benchmarks in the Siemens Suite [6]. Multiple variables within a program in the Siemens Benchmark Suite do not appear to have important relationships that cannot be captured with a single variable instrumentation scheme. The lack of additional improvement in the effectiveness of either ESP or CBI does not justify the decrease in efficiency for these benchmarks.

**ESP vs. OPT.** OPT and ESP perform similarly for the tcas, sched and sched2 benchmarks. In these programs OPT had a score of 90% or higher 40 times while ESP had a score of 90% or higher 38 times. However, for totinfo OPT achieved a score of 85% or higher 19 times while ESP only did so 12 times. ESP’s performance against OPT is encouraging. In three of the four benchmarks in the Siemens Suite ESP approached the maximum effectiveness that is possible for a predicate-based technique using importance score. While ESP offered significant improvements over existing techniques for the program with a large amount of floating-point computations, totinfo, even further elastic predicate improvements are possible.

**ESP vs. Tarantula.** Compared to ESP, Tarantula is not effective for the Siemens Benchmark Suite. For the single variable instrumentation scheme, ESP was able to uniquely identify the statement containing the fault (assign it rank 1) in 14 cases. Tarantula was able to do so in only 3 cases. Even though the ESP approach was able to uniquely identify the faulty statement in 14 cases, only 9 cases yielded scores of 99% or more because in the tcas program the number of statements executed in failing test cases was too few to yield a score of 99%. Tarantula is the only technique in our evaluation that does not consider variable values within an executing program. Instead it only analyzes if a statement was executed. This limits the effectiveness of the technique but enables it to be the most efficient technique evaluated. Efficiency is evaluated further in Subsection D.
C. Sampling and Sparse Test Suites

ESP and CBI both use source code instrumentation to collect program behavior data. The data collection and analysis add overhead to program execution. CBI limits overhead by employing sparse random sampling rather than complete data collection. The sampling is unbiased, collecting a representative subset of the complete program behavior across all runs. To ensure sufficient data collection, CBI relies upon the large user communities of the software for which it is deployed. The result is an effective approach to isolating faults in general-purpose software with wide distribution. ESP is not designed to meet the same goals. In the most common use case, ESP is deployed as a stand-alone fault localization tool for a single SME. In this use case, the goal is to identify failure-predicting predicates as effectively and efficiently as possible for the test cases provided. As a result, complete data collection is often used in favor of sparse random sampling. However, in order to explore the environments in which ESP is useful, it is important to evaluate ESP under sparse random sampling. The goal is to determine the extent to which the uncertainty introduced by sampling reduces the effectiveness of ESP.

The scores of ten executions of each of the 41 programs in totinfo, sched and sched2 for ESP and CBI with sampling rates from 1/10 to 1/10,000 are plotted in Fig. 4. The bottom and top of each box represent the lower and upper quartile and the black band is the median score. The whiskers extend to the lowest and the highest score.

The tcas benchmark is deliberately not included. Recall, each version of tcas contains no loops and less than 100 executed statements. CBI and ESP entail less than one second of overhead, removing the need for sparse sampling. Furthermore, employing sparse sampling on such small executions would not yield significant results.

ESP and CBI are both deployed in the single variable instrumentation scheme. Recall, the score represents the percentage of executed statements in a program a user does not need to examine. For CBI, the sampling is performed in an unbiased manner, as in existing work [3, 4]. Sampling in ESP is more complex. Recall, ESP executes the entire test suite twice. It is executed once to calculate the mean and standard deviation of instrumented program points and a second time to score the elastic predicates generated from the calculated means and standard deviations.

Thus, ESP requires a two-step sampling process. In the first step an estimate of the mean and standard deviation of each program point is computed from profiled values at the sampling rate in an unbiased manner. In the second step, the predicates formed from the estimated mean and standard deviation are scored just as they are in CBI [3, 4].

Fig. 4 shows that the median effectiveness of both ESP and CBI remains stable under sampling rates of 1/10 and 1/100. For less frequent rates, the variance of the scores increases. This is expected given the introduction of random sampling of instrumented program points. At a sampling rate of 1/1,000 both ESP and CBI begin to become less effective. ESP continues to remain more effective than CBI by an absolute margin (~7% source code examined), but the relative difference in effectiveness between the two approaches narrows.

The performance of ESP at a sampling rate of 1/10,000 reveals a trend: sufficiently infrequent sampling rates will reduce the effectiveness of ESP to that of CBI. In these cases the mean and standard deviation of each program point is based on so little data that the resulting elastic predicates are no better, and often worse, than the static and uniform predicates at predicting program failure. However, ESP’s performance under more frequent sampling rates shows that elastic predicates do not require an exact calculation of the mean and standard deviation of values at each program point. Even at infrequent rates like 1/1,000, estimations of the mean and standard deviation result in effective failure-predicting predicates. This is significant, research has shown that sampling rates > 1/1,000 do not necessarily reduce overhead while rates ≤ 1/1,000 do.

Uncertainty in data collection can also be introduced through an incomplete or sparse test suite. Fig. 5 summarizes their effect on ESP and CBI. Test cases from the original test suite for each of the benchmarks in the Siemens Suite were chosen at random, forming sparse test suites at 1/5th, 2/5th, 3/5th and 4/5th the size of the original suite.
Each test case was chosen with uniform random probability without replacement and if the resulting sparse test suite did not contain at least six failing test cases and at least twenty passing test cases it was dissolved and the sparse test suite was reformed. Test suites formed with at least these numbers of passing and failing test cases do not have a strong effect on the effectiveness of existing fault localization techniques in comparison to larger test suites [16]. The scores of ten executions of each of the versions in tcas, totinfo, sched and sched2 for ESP and CBI are plotted. A new sparse test suite was formed for each trial for each test suite size. ESP and CBI are both employed in single variable instrumentation mode.

The effectiveness of ESP and CBI is stable across sparse test suites of different sizes for the benchmarks in the Siemens Suite. Under the sparsest test suite included in the evaluation, 1/5th of the complete test suite, both ESP and CBI show larger variation in their effectiveness scores compared to the other test suite sizes. However, the median effectiveness of each technique at this test suite size is similar to the median of each technique when all test cases are included. Overall, Fig. 5 reveals that for the benchmarks in the Siemens Suite, sparse test suites formed have very little effect on the effectiveness of ESP. In this regard ESP is similar to existing techniques [16].

D. Efficiency

Fig. 6 summarizes efficiency results. Tarantula is the most efficient approach in our study because it only takes into account statement coverage information of passing and failing test cases. All other approaches analyze the values of variables. As a result, we compute the efficiency of each approach relative to Tarantula’s. This relative performance measure is: \( RP = \frac{\text{time}_{\text{tarantula}}}{\text{time}_{\text{other}}} \).

Fig. 6 reveals several trends in the relative performance of the approaches in our evaluation. Each approach improves for tcas and degrades for totinfo. The tcas and totinfo benchmarks are the least and most computationally intensive programs in the suite respectively. Tarantula does not analyze variable computations so it does not reflect these factors. However, the other approaches do and reflect these characteristics of tcas and totinfo.

Given the expected fluctuations, the relative performance of ESP and CBI is independent of the evaluated benchmarks in the Siemens Suite. IVMP is not. For the totinfo benchmark the relative performance of IVMP approaches zero. This drastic degradation is due to several failing totinfo test case executions that IVMP repeatedly re-executes in an attempt to find state alterations resulting in a passing execution. For programs requiring repeated re-executions of failing test cases IVMP is inefficient [13].

E. Widely Used Simulations

We conducted additional experiments to determine how effectively and efficiently ESP localizes faults for the widely used simulations shown in Table 4. The simulations and faults were selected because they either (1) were actual faults in widely used simulations or (2) reflected common faults that have been observed in canonical exploratory software.

The Bates stochastic volatility jump-diffusion pricing model (obtained from [17]) must be calibrated to previous data before it is employed to make price predictions for the future [18]. The model produces an unexpected outcome when the absolute price error of previous data is minimized during calibration instead of the relative price error. The Heston stochastic volatility model (obtained from [19]) is used here to reflect documented issues in the implementation of the logarithm function for complex numbers [20]. The pricing model of European Barrier Options (obtained from [21]) contains a known (and patched) error in computation of bank offering rates. The um-olsr protocol used with the ns2 simulator contains a documented (and patched) error in the degree method [22, 23]. In the 2.19b version of the ns2 network simulator, which we use to model bandwidth for implementations of the TCP protocol, there is a fault, which can incorrectly track the number of nodes in the network [24, 25]. We also seed three simulations built from a widely used simulator with faults [26]. The first is a G/G/1 queueing simulation employing a normal distribution (with infinite tails) when a hump-shaped distribution with values strictly greater than zero is intended [27]. The second is a M/M/c queueing simulation with Poisson distribution implemented incorrectly. The third is a MMPP/D/1 queueing simulation with an incorrectly bound loop.

For each simulation and approach, the rank of the statement containing the fault is shown in Table 4. The RP of ESP and CBI is also provided. For each simulation the best rank is shown in bold. The single variable instrumentation scheme is used for ESP and CBI. Table 4 shows that ESP is capable of significant improvements in effectiveness over the available alternatives. While the decrease in relative performance for ESP does not remain constant when compared to Tarantula, it does not entail more than a 6x decrease compared to CBI. These efficiency results seem reasonable considering: (1) the significant improvement in the rank of faulty statements and (2) the ability to find better failure predicting predicates.

IVMP performs poorly in this portion of the evaluation. This is attributed to two factors: (1) floating-point output from each of the simulations and (2) the extensive use of stochastics within several of the simulations.
The first factor was evident within the Siemens Suite but not to the degree it is here. Recall it is very difficult for IVMP to perform state alterations that cause a failing test case to pass in the totinfo benchmark due to the level of precision in the floating-point computations that generate the program’s output. This causes IVMP to echo Tarantula’s ranking for most statements [13]. All of these simulations also include floating-point output creating the same issue.

The second factor is evident for the simulations that make the most use of stochastic distributions — MC Euro, G/G/1 and M/M/c. In these simulations it is most likely that a test case will pass one time it is executed and fail another time. This creates trouble for a state-altering fault localization approach such as IVMP. IVMP relies on variable values in a failing test case to be replayed and altered to attempt to create a passing test case. However, stochastic distributions cause some variable values to vary from execution to execution. When IVMP attempts to replay and alter a failing execution it is possible for the execution to pass without alteration. We hypothesize these cases affect IVMP’s analysis capabilities. To test our hypothesis we fixed a seed for each of the stochastic distributions used in the three simulations MC Euro, G/G/1 and M/M/c. Given a fixed seed, IVMP is ~2x as effective for the three simulations. However, fixing a seed for each stochastic distribution still does not make IVMP as effective as ESP, and it limits the utility of the simulation to a SME who has likely included the distribution to reflect model uncertainties.

We also evaluated the effectiveness of ESP under sparse sampling rates and sparse test cases. Fig. 7 and 8 summarize the results. For each of the simulations we performed ten trials under various sampling rates and test suite sizes. The rank of the faulty statement for each technique, for each trial is summarized in Fig. 7 and 8 respectively. Given the complexity of the simulations and the ability of infrequent sampling rates to reduce overhead, sampling rates > 1/1000 are not considered [3].

In Fig. 7, under a sampling rate of 1/100,000, both ESP and CBI become significantly less effective and the improvements offered by the elastic predicates in ESP are not as evident and extremely variable. The reductions in effectiveness and variability are due to ESP’s relative poor performance for the Heston, um-olsr and M/M/c simulations compared to the other simulations.

![Figure 7](image1.png)

**Figure 7.** Sampling rates for ESP and CBI. Note the y-axis log scale.

![Figure 8](image2.png)

**Figure 8.** Sparse test suites for ESP and CBI. Note y-axis log scale.

In Fig. 8 the evaluation of the sparse test suites for the simulations is very similar to the evaluation for the Siemens Suite. Both experiments show that the effectiveness of each approach is stable across randomly constructed test suites of varying sizes meeting the characteristics in [16]. The most variation in the effectiveness of ESP and CBI is seen when the test suite is the smallest. For both the Siemens Suite and the widely used simulations, sparse sampling rates reduce the performance of ESP more than sparse test suites.

**F. Multiple Faults**

Several of the versions of the benchmarks in the Siemens Benchmark contain multiple faults. This is not uncommon; simulations or computational models can also contain multiple faults. In each part of our evaluation, each fault localization approach was only required to identify one fault per program. This allowed us to conservatively evaluate the effectiveness of ESP against the best available alternative for the Siemens Suite, IVMP. However, in programs with multiple faults IVMP’s effectiveness can diminish. It is difficult for IVMP to differentiate among multiple faults in a program because it has trouble identifying state alterations in
failing test cases that have different effects on the program output [13]. Modifications to IVMP have been suggested to address this issue but they can result in an even more inefficient approach and do not guarantee the faults are distinguished from one another [14]. The effectiveness and efficiency of ESP and CBI do not diminish in the face of multiple faults. Both approaches guarantee a failure-predicting predicate for each fault in a program [3].

G. Validity

Internal, external, and construct validation threats affect our evaluation. Threats to internal validity arise when factors affect the dependent variables without the evaluators' knowledge. It is possible that some implementation flaws could have affected the results. However, our results for the Siemens Suite benchmarks for IVMP and Tarantula closely matches results from previous work [6, 13, 15]. Threats to external validity occur when the results of our evaluation cannot be generalized. Although we performed our evaluations on 12 subjects with a total of 90 versions, we cannot claim that the effectiveness of ESP can be generalized to other faults in other software subjects. However, the results for the 90 versions demonstrate the power of elastic predicates in comparison to static and uniform predicates. Threats to construct validity concern the appropriateness of the metrics used in our evaluation. More studies into how useful developers find statement-ranking metrics need to be performed. However, the more accurate fault-localization methods are the more meaningful such studies will become.

IV. RELATED WORK

There is no body of literature that specifically addresses the analysis of exploratory software outcomes. Statistical fault localization techniques come the closest to addressing the need. Tarantula [15] and CBI [3, 4] are described in Section 3. SOBER [6] uses the concept of evaluation bias to express the probability that a predicate is evaluated to be true in an execution. BARINEL pairs abstractions of program traces with Bayesian reasoning to deduce multiple-fault candidates and their probabilities [28].

Within different techniques, different metrics are employed to rank statements. These metrics include importance [3], increase [3], F-1 measure [29], Tarantula's suspiciousness [15], capture propagation [30] and the Ochiai metric [16]. Baah et al. have shown that the effectiveness of each of these techniques can be also improved when the ranking metric is integrated with a cause-effect metric [29, 31]. In future work we will explore the extent to which these improvements in effectiveness and improvements from elastic predicates are orthogonal.

Additional enhancements exist for these techniques. Adaptive Bug Isolation significantly reduces the number of program points that need to be instrumented to identify predicates with high importance scores [32]. This improves overall efficiency but not effectiveness. Tarantula can be extended to facilitate multiple developers debug a faulty program in parallel [33]. Tarantula has been further enhanced to identify groups of statements that are always executed by a subset of the original test suite [34]. CBI has been adapted to handle compound Boolean expressions [35]. In Holmes, Chilimbi et al. conduct statistical fault localization using paths instead of predicates [36]. The profiling of paths in Holmes can be viewed as informal elastic predicates. Zhang et al. show that short-circuit rules in the evaluation of Boolean expressions may significantly affect the effectiveness of predicate-based techniques [37]. Santelices et al. show that the integrated results of fault localization techniques using different program entities may be better than the use of any single program entity [38].

Techniques to link failures to inputs exist as well. PENUMBRA marks program inputs as they enter an application and tracks them during execution to identify test cases relevant to observed failures [39]. Snap performs similar analysis and identifies critical regions of code related to observed failures [40]. Artzi et al. offer techniques to generate smaller test suites than standard techniques with the same fault-localization characteristics [41].

State-altering approaches take an experimental approach to fault localization. The Delta Debugging framework computes cause-effect chains linking failure-inducing input to faulty statements [42]. Predicate switching attempts to isolate faulty statements by identifying predicates whose outcomes can be altered during a failing test case to cause the test case to pass [8]. IVMP, described in Section 3, is the most effective state-altering approach for the benchmarks in the Siemens Suite [13, 14]. These approaches can be inefficient because the subject program must be re-executed multiple times and the consistency of alterations must be ensured. Furthermore, our evaluation suggests that stochastic distributions within subject programs can degrade analysis.

Slicing is another means of locating faults. Gupta et al. narrow down slices using a forward dynamic slicing approach [44], while Zhang et al. [8] integrate forward and backward dynamic slicing approaches for debugging. Instrumentation and sampling approaches for continuous path and edge profiling that can be used for fault localization also exist [45]. While these approaches have no direct bearing on the direction of ESP, they do suggest potentially interesting hybrid approaches to explore in future work.

V. CONCLUSION

SMEs can struggle for decades with separating valid, but unexpected, exploratory software outcomes from failures. This remains an open problem. However, we have developed a predicate-based statistical debugging method, ESP, which localizes sources of unexpected outcomes. ESP complements the predicates used in existing approaches with elastic predicates. The result is improved effectiveness. ESP outperforms the best alternatives in widely used simulations and performs as well as the best alternative (IVMP) for fault localization benchmarks. These improvements persist in the face of sparse test suites and sparse sampling rates.

REFERENCES


[22] UM-OLSR. http://maximum.dif.um.es/~Software/UM-OLSR.


