Statistical Debugging with Elastic Predicates

Ross Gore, Paul F. Reynolds, Jr.
Dept. of Computer Science
University of Virginia
Charlottesville, USA
{rjg7v,pfr}@virginia.edu

David Kamensky
Institute of Computational Engineering and Sciences
University of Texas at Austin
Austin, USA
kamensky@ices.utexas.edu

***Abstract*—Traditional debugging and fault localization methods have addressed localization of sources of software failures. 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 introduce profiling of values assigned to variables within a failing program. These elastic predicates are better predictors of software failure than the static and uniform predicates used in existing techniques such as Cooperative Bug Isolation (CBI). We present experimental results for established fault localization benchmarks and widely used simulations that show improved effectiveness.**

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 the floating-point computations that generally accompany them. Predictions based on exploratory software outcomes have entered the mainstream of critical public policy and research decision-making practices, affecting large numbers of people and valuable resources.

Unfortunately subject matter experts (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 with elastic predicates. ESP and elastic predicates are novel, and extend the domain of programs for which fault localization analysis has been shown to be effective. They are explained further in the remainder of this paper.

II. ELASTIC PREDICATES AND ESP

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 represents a class of fault localization techniques that share a common structure. Each consists of a set of conditional propositions, or predicates, inserted into a failing program and tested at particular program points. The predicates are given an importance score based on how frequently they are true in passing and failing test cases, ranked and provided to SMEs to assist in finding and fixing faults.

In the canonical predicate-based statistical debugger Cooperative Bug Isolation (CBI), three predicates are tested for each assignment statement to, or return of, a variable x: (x>0), (x=0) and (x<0) [1]. These predicates are *uniform* and *static*. They are *uniform* in the sense that for each assignment to, or return of, a variable x, the same set of conditional propositions partition the values of x in the same manner. The predicates are *static* because each conditional proposition is determined before the execution of the program; the predicates do not change based on dynamic analysis of the values of variable x. Within ESP these three static and uniform predicates are complemented with *elastic predicates*. Elastic predicates use dynamic analysis to compute the mean, μ_x, and standard deviation, σ_x, of the values assigned to, or returned from, a variable x.

Computing elastic predicates in ESP is a two-step process. First, as ESP executes test cases, the mean, μ_x, and standard deviation, σ_x, for an assignment to, or return of, a variable x are computed using online algorithms requiring constant space. Once all test cases are executed, each μ_x and σ_x is stored. Then, each μ_x and σ_x is used to insert nine elastic predicates at each statement or return. These predicates partition the values for an instrumented program point for three standard deviations above and below μ_x. They are defined below, where i = 0, 1, 2.

- $x < \left(\mu_x - 3\sigma_x\right)$ and $x > \left(\mu_x + 3\sigma_x\right)$

978-1-4577-1639-3/11/$26.00 © 2011 IEEE ASE 2011, Lawrence, KS, USA

- $\mu_x = x$
- $\left(\mu_x - (i+1)\sigma_x\right) \le x < \left(\mu_x - (i\sigma_x)\right)$ and $\left(\mu_x + i\sigma_x\right) < x \le \left(\mu_x + (i+1)\sigma_x\right)$

In the second step, the test cases are re-executed and the inserted predicates are assigned importance scores. Subsection *A* describes scoring further. Subsection *B* presents an example where elastic predicates are employed.

A. Importance Scores

The extent to which a predicate predicts program failure 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 an importance score: a one bit feedback report, *R*, indicating if the test case passed or failed and a vector *V* with a one bit entry for each predicate. Within *V*, each entry indicates if the corresponding predicate is observed to be true during test case execution. The data for predicate *p* from each feedback report *R* and each corresponding *V* is aggregated into these measures to account for *specificity* (also called *precision*) and *sensitivity* (also called *recall*) [1]:

1. $S(p\ obs)$ and $F(p\ obs)$ - the number of successful and failed test cases where *p* was evaluated.
2. $S(p)$ and $F(p)$ - the number of successful and failed test cases where *p* was evaluated and was true.

Increase uses these measures to estimate specificity. This estimate accounts for the amount that *p* being true increases the probability of failure, over reaching where *p* is evaluated.

$$Increase(p) = \frac{F(p)}{S(p)+F(p)} + \frac{F(p\ obs)}{S(p\ obs)+F(p\ obs)} \quad (1)$$

Sensitivity is estimated by the quantity: $\log(F(p))/\log(totalFailed)$. *totalFailed* is the number of failing test cases. This estimate accounts for the number of failing test cases in which a predicate is observed. The importance score for a predicate *p* combines sensitivity and specificity.

$$Importance(p) = \frac{2}{\frac{1}{Increase(p)} + \frac{1}{\log(F(p))/\log(totalFailed)}} \quad (2)$$

B. BC Example: Elastic vs. Static and Uniform Predicates

Fig. 1 shows the `more_arrays()` function in the 1.06 version of BC, a basic command-line calculator tool [2]. The `more_arrays()` function is responsible for increasing the number of arrays needed for computing. 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 extra space. However, there is a fault. 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 from `old_count` through `v_count - 1`. Here, the new storage buffer has room for `a_count` elements, but the second loop is incorrectly bound by `v_count`, so when `v_count > a_count` the program fails. The three static and uniform predicates employed in CBI do not make the fault easy to discern. Most values assigned to the variables in `more_arrays()` are greater than zero, and satisfy the same predicate, (x>0). This predicate does not closely match where the fault is expressed, yielding a low importance score. However, the elastic predicate, `indx` > $\mu + 3\sigma$, yields a high importance score because it clusters together unusually large values for `indx`. The predicate captures the fault: when the input to BC defines an unusually large number of arrays, the program fails. [1].

```
152 void
153 more_arrays()
154 {
155   int indx;
156   int old_count;
157   bc_var_array **old_ary;
158   char **old_names;
159
160   /* Save the old values. */
161   old_count = a_count;
162   old_ary = arrays;
163   old_names = a_names;
164
165   /* Increment by a fixed amount and allocate. */
166   a_count += STORE_INCR;
167   arrays = (bc_var_array **) bc_malloc (a_count*sizeof(bc_var..
168   a_names = (char **) bc_malloc (a_count*sizeof(char *));
169
170   /* Copy the old arrays. */
171   for (indx =1; indx < old_count; indx++)
172     arrays[indx] = old_ary[indx];
173
174
175   /* Initialize the new elements. */
176   for (; indx < v_count; indx++)
177     arrays[indx] = NULL;
178
179   /* Free the old elements. */
180   if (old_count != 0)
181     {
182       free (old_ary);
183       free (old_names);
184     }
185 }
```

Figure 1. The souce code of the more_arrays() function in BC.

III. EVALUATION

A. Experimental Setup

The utility of a fault localization technique is determined through experimental evaluation. In our evaluation, we employ (1) a set of widely used simulations which make extensive use of stochastics and floating point computations and (2) a subset of the Siemens Benchmark Suite [3]. The included Siemens benchmarks are: `tcas`, `totinfo`, `schedule` and `schedule2`. These were chosen because (1) they contain a large amount of floating-point computations or (2) are a simulation. The Siemens benchmarks test ESP's effectiveness with well-known applications in the fault localization community. However, the widely used simulations better reflect our target application domain due to their larger size and extensive use of stochastics and floating-point computations. The evaluation includes the following ranking approaches:

ESP. ESP ranks statements by 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 effectiveness from an elastic predicate-based approach.

CBI. CBI uses the three static and uniform predicates previously described. To ensure fairness CBI is extended to consider floating-point type variables. It uses ESP ranking.

Tarantula. Tarantula ranks statements according to the suspiciousness (*susp*) of a statement *s* [5]:

$$susp(s) = \frac{failed(s)}{(failed(s)/totalFailed) + (passed(s)/totalPassed)} \quad (3)$$

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

IVMP. Interesting Value Map Pairs (IVMP) ranks each statement based on the number of failing executions where a state alteration creates a passing execution [4]. The Tarantula ranking system is used to break ties.

B. Widely Used Simulations

We conducted effectiveness experiments for the widely used simulations shown in Table I. Here we describe each simulation. Due to the size and complexity of the simulations OPT is not included in this portion of the evaluation. The Bates stochastic volatility jump-diffusion pricing model [6] must be calibrated to previous data before it is employed to make price predictions for the future [7]. The model produces an unexpected outcome when *relative price error* of previous data, is minimized during calibration, instead of the *absolute price error*. The Heston stochastic volatility model [8] is used here to reflect documented issues in the implementation of the logarithm function for complex numbers [9]. The pricing model of European Barrier Options contains a known error in computation of bank offering rates [10]. The um-olsr protocol used with the ns2 network simulator contains a documented error in the degree method [11]. In the 2.19b version of the ns2 network simulator, there is a fault, which can incorrectly track the number of nodes in the network [12, 13]. We also seed faults in three queueing simulations built from a widely used simulator [14]. The first is a G/G/1 simulation employing a normal distribution when a hump-shaped distribution with values strictly greater than zero is intended. The second is a M/M/c simulation with a Poisson distribution implemented incorrectly. The third is a MMPP/D/1 simulation with an incorrectly bound loop.

For each simulation and approach, the rank of the statement containing the fault is shown in Table I. The best rank for each simulation is bolded. The table shows that ESP is capable of significant improvements, at times approaching an order or magnitude improvement. However for these simulations, IVMP performs poorly because of: (1) floating-point output and (2) the extensive use of stochastics within several simulations. The first factor is evident in all the simulations. For programs with floating-point output it is difficult for IVMP to perform state alterations that cause a failing test case to pass. This is due to the level of precision in the floating-point computations that generate the simulation output. Thus, the approach echoes Tarantula's rankings for most statements [4].

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. We hypothesize that these cases affect the analysis capabilities of a state altering approach like IVMP. To test this we fixed a seed for each of the stochastic distributions used in the three simulations. Given a fixed seed, IVMP is ~2x as effective. However, fixing a seed still does not make IVMP outperform ESP, and it limits the utility of the simulation to a SME.

TABLE I. EFFECTIVENESS RESULTS (BOLD SCORES ARE BEST)

Name	ESP	CBI	IVMP	Tarantula
Bates	**3**	56	42	78
Heston	**1**	16	68	144
MC Euro	**5**	38	35	45
um-olsr	**23**	196	87	268
ns2 v2.19b	**21**	107	145	241
G/G/1	**1**	**1**	296	314
M/M/c	**12**	68	157	157
MMPP/D/1	**13**	96	64	213

C. Siemens Suite Benchmarks

Next, we evaluate the approaches using the Siemens Benchmark Suite. We assign a score to each set of statements that is the percentage of executed statements that need not be examined. Given a ranked list of statements *S*, with the faulty statement at rank *r*, the score is: $score(S) = (|S|-r)/|S| * 100$. Fig. 2 shows our results. The x-axis represents the lower bound of each score range, and the y-axis represents the percentage of versions with a score greater than the lower bound.

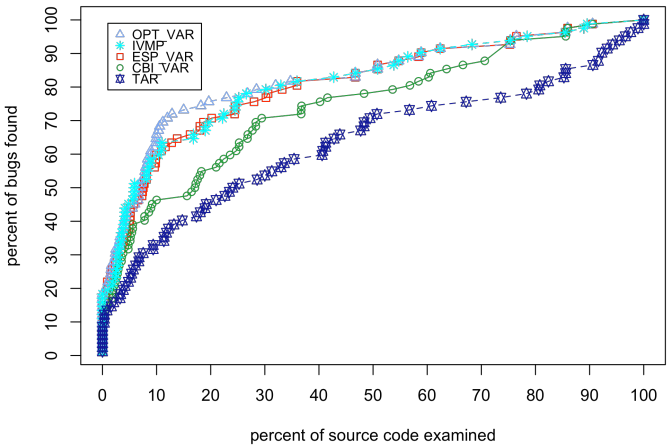

Figure 2. Ranking approaches for the Siemens Benchmark Suite.

ESP vs. CBI. ESP outperforms CBI. This is to be expected, ESP offers the same predicates as CBI along with the nine elastic predicates resulting in a better tool for identifying predicates with high importance scores. Higher importance scores leads to more effective fault localization.

ESP vs. OPT. OPT and ESP perform similarly for the `tcas`, `sched` and `sched2` benchmarks. 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. While ESP offered similar effectiveness for three of the four benchmarks, the better elastic predicates in OPT enable even further improvements for programs with large amounts of floating-point computations like `totinfo`.

ESP vs. Tarantula. Tarantula is not as effective as ESP for the Siemens Benchmark Suite. Tarantula is the only technique in our evaluation that does not consider variable values. This limits the effectiveness of Tarantula but makes it the most efficient technique evaluated.

ESP vs. IVMP. 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 such a score 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. Recall, that this difficulty is due to the level of precision in the floating-point computations that generate the program's output [4]. As a result, IVMP performs the same as or worse than Tarantula for 15 of the 23 versions of `totinfo`.

IV. Related Work

Existing statistical fault localization techniques come the closest to addressing our need of analyzing exploratory software outcomes. Tarantula [5] and CBI [1] are described in Section 3. SOBER [15] uses the concept of evaluation bias to express the probability that a predicate is true in an execution. BARINEL pairs abstractions of program traces with Bayesian reasoning [16]. Within different techniques, different metrics are employed to rank statements: importance [1], increase [1], F_1 measure [17], Tarantula's suspiciousness [5], capture propagation [18] and the Ochiai metric [17, 18]. The effectiveness of each of these can be improved when the ranking metric is integrated with a cause-effect metric [17]. Additional enhancements exist. CBI can employ compound Boolean expressions to improve effectiveness and adaptive sampling can improve efficiency [19, 20]. Paths (serving as informal elastic predicates) can be used instead of predicates to offer improved effectiveness [21]. Short-circuit rules in the evaluation of Boolean expressions can also improve effectiveness [22]. Santelices et al. show that integrated results with various program entities can offer improvements in effectiveness over using any single program entity [23].

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 statistical debugger, ESP, which localizes sources of unexpected outcomes. ESP complements the predicates used in existing approaches with elastic predicates to improve effectiveness. ESP outperforms the best alternatives in widely used simulations and performs as well as the best alternative for established benchmarks.

References

[1] B. Liblit, "Cooperative Bug Isolation (Winning Thesis of the 2005 ACM Doctoral Dissertation Competition)," Lecture Notes in Computer Science vol. 4440, 2007.

[2] BC Calculator. http://www.gnu.org/software/bc/

[3] SIR: Software-artifact Infrastructure Repository. http://sir.unl.edu/portal/index.html.

[4] D. Jeffery, N. Gupta and R. Gupta, "Fault Localization Using Value Replacement," Proc. of Int. Symp. Soft. Testing and Analysis, ACM Press, July, 2008, pp. 167–177.

[5] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," Proc. of Int. Conf. on Automated Soft. Engineering, ACM Press, Nov. 2005, pp. 273–282.

[6] Financial Quantitative Algorithms. http://www.javaquant.net/

[7] K . Detlefsen and W. K . Härdle, "Calibration Risk for Exotic Options," Journal of Derivatives vol. 14, Aug. 2007, pp. 47-63.

[8] Directory of Open Source Software for Quantitative Finance & Trading. http://www.quantcode.com

[9] S. Mikhailov and U. Nögel. Heston's Stochastic Volatilty Model: Implementation, Calibration, and some Extensions. Wilmott, July 2003, pp. 74–79.

[10] QuantLib. http://quantlib.org/index.shtml

[11] Bug in um-olsr? http://stackoverflow.com/questions/4190561/bug-in-um-olsr-for-ns-2-34

[12] ns-2: The Network Simulator. http://is.edu/nsam/ns

[13] ns2 Problems. http://www.isi.edu/nsnam/ns/ns-problems.html

[14] Simpack Toolkit. http://cise.ufl.edu/~fishwick/simpack.html

[15] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," Proc. of Symp. on Foundations of Soft. Engineering, ACM Press, Sep. 2005, pp. 286–295.

[16] R. Abreu, P. Zoeteweij, and A. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," In Proc. of Testing: Acad. and Ind. Conf. Prac. and Res. Tech., IEEE Press, Sept. 2007, pp. 89–98.

[17] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal Inference for Statistical Fault Localization," In Proc. of Int. Symp. on Software Testing and Analysis, ACM Press, July 2010, pp. 73-83.

[18] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang and X. Wang. "Capturing Propagation of Infected Program States," Proc. of Symp. on Foundations of Soft. Engineering, ACM Press, Sep. 2009, pp. 43-52.

[19] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," Proc. of Int. Conf. on Soft. Engineering, ACM Press, May 2010, pp. 255- 264.

[20] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound Boolean predicates," Proc. of Int. Symp. on Soft.Testing and Analysis, ACM Press, July 2007, pp. 5–15.

[21] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," Proc. of Int. Conf. Soft. Engineering, ACM Press, May 2009, pp. 34–44.

[22] Z. Zhang, W. Chan, T. Tse, P. Hu, and X. Wang, "Is non- parametric hypothesis testing model robust for statistical fault localization?" Info. and Soft. Tech. vol. 51, Nov. 2009, pp. 1573-1585.

[23] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," Proc. of Int. Conf. on Soft. Engineering, ACM Press, May 2009, pp. 56-66.